UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Diseño e implementación de un framework para la monitorización de aplicaciones usando OpenFlow

Máster Universitario en Ingeniería de Telecomunicación

Autor: ESPARZA JUANDEABURRE, Alejandro
Tutor: RAMOS DE SANTIAGO, Javier
Ponente: LÓPEZ DE VERGARA MÉNDEZ, Jorge E.

Junio 2020

Diseño e implementación de un framework para la monitorización de aplicaciones usando OpenFlow

AUTOR: Alejandro Esparza Juandeaburre
TUTOR: Javier Ramos de Santiago
PONENTE: Jorge E. López de Vergara Méndez

High Performance Computing and Networking
Dpto. Tecnología Electrónica y de las Comunicaciones.
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2020

Resumen

Vivimos en una época en la que la tecnología avanza rápidamente, por lo que, saber adaptarse y evolucionar conforme se van produciendo los cambios es vital, tanto para grandes empresas, como en entornos educativos con el fin de tener la capacidad de formar a las personas en consonancia hacía dónde están derivando las redes. Uno de los cambios más relevantes en los últimos años ha sido la implantación de redes SDN, redes definidas por software.

Este trabajo de fin de Máster consiste en el desarrollo e implementación de un *framework* de monitorización de tráfico a nivel de aplicación para las redes SDN. En la actualidad existen gran cantidad de fabricantes que apuestan por dicha tecnología y, en este caso, el estudio se va a basar en el uso del protocolo *OpenFlow*. En este protocolo, por defecto, la monitorización a nivel de la capa de aplicación no está disponible lo cual limita el tipo de monitorización y estadísticas que podemos obtener. Sin embargo, el protocolo *Openflow* sí que permite la obtención de estadísticas (bytes y paquetes) para flujos definidos por campos de los niveles Ethernet, IP y TCP entre otros. La idea que se propone en este trabajo es aprovechar ese sistema de monitorización ya existente y construir sobre el mismo una capa de abstracción que nos permita la obtención de estadísticas (bytes y paquetes por flujo) relacionadas con la navegación Web focalizándonos en el análisis de HTTP, SSL y DNS. Estas estadísticas que obtenemos de *Openflow* se enriquecen, además, con información del dominio o host extraídos de la inspección del tráfico que realiza el *framework* lo cual aporta un grano más fino a la hora de dimensionar y analizar la información.

Para abordar la tarea de este trabajo, en primer lugar, se realizará un breve repaso sobre las redes SDN y sus aplicaciones, después, se implementará el *framework* de monitorización utilizando los lenguajes C y Python.

Una vez implementado el *framework* se realizarán pruebas funcionales y de rendimiento en una topología controlada formada por tres *hosts*, un *switch* y un controlador SDN para analizar los límites de funcionamiento del sistema desarrollado. También se realizarán pruebas de validación comparando las estadísticas y datos extraídos del *framework* con datos obtenidos con *tshark* que usaremos como *ground truth*. Por último, se mostrará la integración del *framework* con un sistema de representación de datos como es Grafana.

Palabras clave

SDN, Open source, OpenFlow, framework, switch, host, controlador, monitorización, Python, C, TCP, IP, ground truth, tshark, DNS, HTTP, SSL.

Abstract

Nowadays, technology is progressing quickly so, knowing how to adapt and evolve to it while the changes are taking effect, it is crucial. The enterprises and the educational departments need to have the ability to develop their people according where the new features of the network are going. One of the biggest changes in networks it is SDN, Software Defined Networks.

This Master's Thesis is about developing and implementing a monitoring framework for the emerging SDN networks at the application layer. To achieve the main goal, many different manufacturers have bet about monitoring, but this Thesis is based in OpenFlow software, which is open source. By default, the monitoring layer it is not developed, fact that limit the monitoring and statistics types that we can get. However OpenFlow protocol, allows obtaining statistics (bytes and packets) per flows defined by fields at the Ethernet, IP and TCP levels. The goal proposed in this Thesis is to take advantage of this existing monitoring system and build on it an abstraction layer that allows us to obtain statistics (bytes and packets per stream) related to Web browsing, focusing on HTTP, SSL and DNS. The statistics that we obtain from OpenFlow are also enriched with information from the domain or host extracted from the traffic inspection carried out by the framework, which provides a finer grain when it comes to dimensioning and analyzing the information.

To deal with the task of this work, first, a brief review will be carried out of SDN networks and their applications, and then the monitoring framework will be implemented using the C and Python languages.

Once the framework is implemented, functional and performance tests will be accomplished in a controlled topology consisting of three hosts, a switch and an SDN controller to analyze the operating limits of the developed system. Validation tests will also be performed comparing the statistics and data extracted from the framework with data obtained with tshark that we would use as ground truth. Finally, the integration of the framework with a data representation system such as Grafana will be shown.

Keywords

SDN, Open source, OpenFlow, framework, switch, host, controller, monitor, Python, C, TCP IP, ground truth, tshark, DNS, HTTP, SSL.

Agradecimientos

A todos quienes me han acompañado durante este trayecto.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Motivación

Debido a la gran sofisticación y los constantes cambios a los que están sometidos actualmente las redes de comunicaciones las tareas de gestión, monitorización y operación se convierten en labores cada vez más complicadas de abordar. Por ello, la búsqueda de protocolos y mecanismos que faciliten dichas tareas es de vital importancia para los administradores de las redes. En la actualidad, existen nuevas tecnologías que buscan facilitar todas esas tareas de gestión, monitorización y operación de la red, de ahí su especial potencial e importancia. Este trabajo de fin de Máster se va a centrar en las redes definidas por Software, que surgieron para dar solución a los problemas de gestión de las redes, entre otros.

Las redes definidas por Software, a partir de ahora denominadas redes SDN [1], introducen un nuevo enfoque en la arquitectura de red, el cual aporta una serie de ventajas a las aproximaciones tradicionales. SDN, hace la red, desde el punto de vista técnico, más ágil, manejable y desde el punto de vista económico, más rentable. Es capaz de adaptarse a la velocidad y volumen de datos que atraviesan las redes de los proveedores de internet que con el paso del tiempo se va incrementando (siempre hay que tener en cuenta el hardware) y ofrece una solución muy versátil a los administradores de la red.

SDN se puede definir, en su acepción más básica, como la separación entre las funciones del plano de control y las funciones de reenvío de datos, simplificando la definición aún más, separación entre plano de control y plano de datos. SDN, en definitiva, centraliza la inteligencia de la red y abstrae la arquitectura subyacente de las aplicaciones y servicios. El software de control aporta la capacidad de operar y ajustar la red, lo que en definitiva permite una orquestación [2] inteligente.

Uno de los protocolos que dieron comienzo a esta solución y ahora está muy extendido en el despliegue de las redes SDN, es OpenFlow [3]. La motivación del trabajo de fin de Máster surge a raíz de la necesidad de implementar un sistema de monitorización de la capa de aplicación utilizando como base las funcionalidades de monitorización que ya proporciona el protocolo OpenFlow. Para ello, es necesario comprenderlo, y así poder explotar y aprovechar al máximo todas las funcionalidades que puede ofrecer en el despliegue de redes SDN, centrándonos sobre todo en el aspecto de la monitorización. En el estudio se va utilizar tráfico variado para las pruebas. El tráfico que se va a utilizar es DNS, HTTPs y HTTP. OpenFlow presenta una limitación en tanto en cuanto, los campos que hacen match están restringidos a la cabecera de los paquetes únicamente. Por ello, surge la necesidad de la inspección de los paquetes y de la carga útil para poder distinguir flujos y poder aplicar reglas acordes a las necesidades.

El trabajo de fin de Máster trata de aprovechar la programabilidad de las redes SDN mediante OpenFlow, por ello se va a desarrollar e implementar un *framework* que permita monitorizar las aplicaciones utilizando OpenFlow. La necesidad de llegar hasta la capa de aplicación viene porque el protocolo OpenFlow actualmente permite una monitorización desde nivel de capa física hasta la capa de transporte. Una vez que se implemente el *framework* de monitorización se podrán automatizar reglas sobre la red para favorecer y optimizar el funcionamiento de la red y sus aplicaciones.

El encargado de gestionar y dirigir la red SDN, se denomina controlador. Es quien aporta inteligencia y dónde se definen las políticas y reglas que se aplicaran sobre los distintos flujos de tráfico que atraviesan la red.

Existe un amplio y variado espectro de controladores en las redes SDN y, es necesario tener constancia de ellos y realizar una elección correcta ya que hay algunos que funcionan mejor en unos entornos u otros. Para nuestro estudio será más adecuados controladores ligeros con capacidades para monitorización ya que lo que se pretende es construir una maqueta rápida para evaluar la viabilidad de la solución. Si se requiere llevar a producción la solución sería necesario realizar un estudio similar y seleccionar otros controladores para determinar cuál puede tener cabida.

Para nuestro proyecto, se ha optado por el controlador RYU ya que encaja a la perfección con las necesidades del estudio. A grandes rasgos, RYU, es un controlador desarrollado en Python que soporta OpenFlow (también soporta NETCONF [4] y OF-config [5]) y dispone de una API REST que lo ayuda a integrarse con otros sistemas además de ser apto para la monitorización de la red.

Como última motivación del trabajo se encuentra hacer uso de herramientas que permitan emular redes que posteriormente pueden ser desplegadas en la vida real. La capacidad de realizar maquetas puede ayudar en gran medida antes de desplegar los servicios en la red de producción ya que es posible detectar posibles fallos y corregirlos con antelación y que no causen daños en la red reduciendo los costes. Para el caso de estudio que nos atañe se va a utilizar Mininet [6], para simular la red SDN.

Una vez estudiados los elementos que influyen en el trabajo se realizara una serie de pruebas y medidas que se compararan para poder determinar cómo de bueno es el *framework* implementado, con respecto a los resultados obtenidos con tshark [7] utilizados como *ground truth*.

1.2 Objetivos

El objetivo principal de este trabajo es el diseño e implementación de un sistema de monitorización a nivel de aplicación usando *OpenFlow*. Entrando más en detalle y para no hablar de una manera tan generalizada, los objetivos contenidos bajo el propósito principal son, (i) el manejo de software simulación de redes definidas por software, (ii) el desarrollo de un *sniffer* que haga DPI sobre el tráfico y (iii) envío de los datos capturados hacia el controlador, lugar donde se recoge toda la información y se ha implementado un colector que agrupa por flujos y da sentido a dichos datos. Como último objetivo y para determinar el rendimiento del sistema de monitorización se representan los flujos obtenidos y se comparan con datos obtenidos mediante *tshark* para poder dirimir cómo de bueno es el sistema de monitorización implementado.

En resumen, los principales objetivos son:

- Investigar los principales controladores *open source* del mercado para poder determinar cuál se adecúa más al estudio.

- Desplegar del entorno de pruebas mediante el programa Mininet. Base sobre la que posteriormente se desarrollaran las pruebas.

- Implementar del programa encargado de recibir los paquetes un puerto de *SPAN* [8] y el posterior envío al controlador para ser procesados.

- Implementar el programa encargado de procesar los paquetes recibidos del *sniffer*.

- Representar los datos procesados para poder compararlos con medidas obtenidas como *ground truth*

- Analizar y evaluar los resultados obtenidos para determinar el rendimiento del *framework* desarrollado.

1.3 Planificación

El desarrollo del *framework* de monitorización conlleva un trabajo previo que comienza con el estudio de los elementos que van a estar involucrados a lo largo de todo el trabajo y de los que hay que conocer sus principales características. Una vez que se disponga de una visión más amplia de cada componente que va a formar parte del diseño se puede comenzar a desplegar el escenario sobre el que se realizará el estudio.

Tras el estudio previo de las distintas alternativas sobre los elementos que pueden tener cabida y la toma de decisión de cuáles van a formar parte del trabajo de fin de máster, se procede a la implementación y desarrollo de los componentes claves para el funcionamiento del sistema de monitorización. La implementación está basada en dos grandes módulos.

El primero de los módulos implementados es el *sniffer* utilizando las librerías de pcap [9]. El segundo módulo de implementación se corresponde con el colector.

Tras la implementación se puede proceder con la ejecución de las pruebas y la representación de las mismas.

Cuando se obtienen los resultados uno de los últimos pasos es representarlos gráficamente para analizar el comportamiento y evaluar el rendimiento del *framework*. Durante esta fase se escogió el software de representación de los datos. El software escogido es Grafana [10]. En paralelo. se selecciona la base datos de la cual después se nutrirá el software de representación para mostrar los resultados.

El último paso en la realización del trabajo de fin de Máster es el análisis y estudio de los resultados obtenidos y con ello la obtención de conclusiones finales.

Las fases esquemáticamente se pueden desglosar en las siguientes:

- Documentación: Esta fase abarca todo el proceso de lectura y recopilación de información y *papers* de divulgación científica, con el fin de obtener las herramientas que más se ajustan al estudio realizado.

- Diseño: Fase que cubre el escenario, y el papel que va a desempeñar cada elemento de la red. Se toman decisiones sobre la conectividad entre el *sniffer* y el colector del controlador.

- Implementación: Se desarrollan los programas en C y Python para el correcto funcionamiento del escenario.

- Pruebas: Una vez que esta todo implementado todo el escenario se verifica el funcionamiento y se realizan varios tipos de prueba para obtener graficas de interés para el análisis.

- Análisis de resultados: A través de las pruebas realizadas y comparando con resultados obtenidos de manera objetiva con un software considerado como el estándar *de facto* tanto en la industria como en el mundo de la investigación.

- Obtención de conclusiones: Tras analizar los resultados se valora si se han alcanzado los objetivos iniciales y se evalúa de manera global el proyecto.

Con el fin de resumir las tareas anteriormente mencionadas se incluye el siguiente diagrama de Gantt:

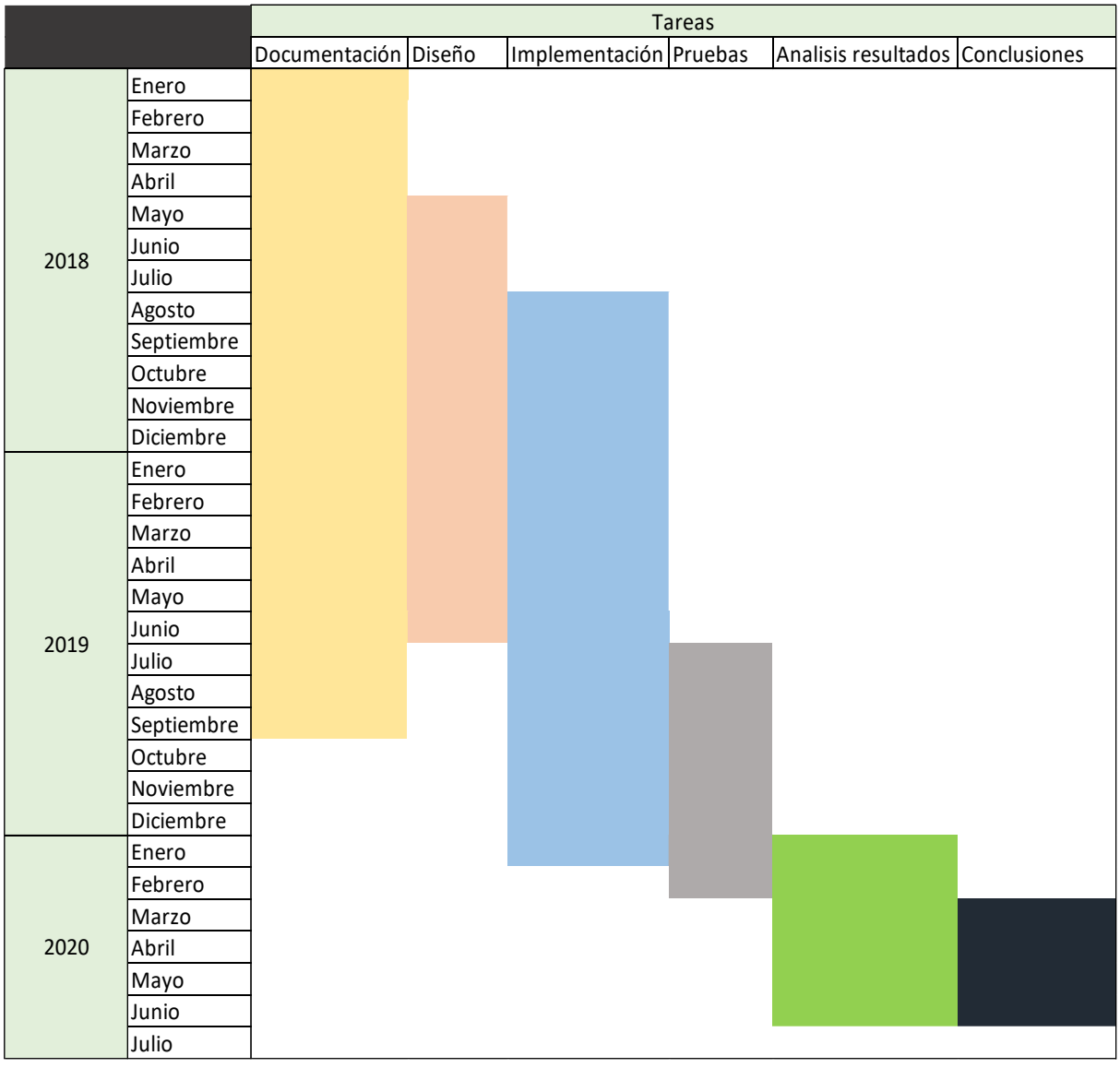


Figura 1-1: Cronograma de realización del Trabajo de Fin de Máster

Pese a que en el diagrama aparezca muy dilatado en el tiempo, se han invertido las 300h estipuladas para la realización del trabajo de Fin de Máster.

1.4 Organización de la memoria

La memoria está estructurada en los siguientes capítulos:

- **Capítulo 2.** En este capítulo se hace un repaso sobre el estado del arte. Se definen los principales conceptos en los que se basa el Trabajo de Fin de Máster, como puede ser *OpenFlow*, el estado de la monitorización en redes SDN, sus tipos de controladores y tipos de protocolos disponibles para desplegarlo.

- **Capítulo 3.** En él se detalla el diseño que va a servir como base para la implementación, así como también, las decisiones tomadas para su implementación.

- **Capítulo 4.** En el cuarto capítulo se explica el proceso a alto nivel para su desarrollo de los distintos elementos del *framework*, y también los métodos para la inyección de tráfico en la red.

- **Capítulo 5.** Es el capítulo donde se pone a prueba el *framework* implementado. Las principales pruebas a las que se ve sometido es de rendimiento, para los distintos tipos de tráfico. Pruebas sobre el error relativo producido y una visión global final.

- **Capítulo 6.** En último lugar se describen las conclusiones extraídas de las pruebas realizadas y sobre el trabajo en conjunto y se proponen líneas de trabajo para mejorar y ampliar el alcance del actual.

2 Estado del arte

2.1 ***Introducción***

Se va a repasar en qué situación se encuentran los principales temas que han sido objeto de estudio durante el desarrollo del trabajo de fin de Máster. Para comenzar se aportará una visión sobre lo que es SDN y sus arquitecturas, en siguiente lugar, se destacarán las principales maneras de desplegar redes SDN mediante los distintos protocolos, haciendo mayor énfasis en *OpenFlow*, ya que es el protocolo presente en el desarrollo del actual estudio. Es muy importante tener una visión de la evolución de *OpenFlow*, ya que fue el pionero en las redes definidas por software, y es necesario ver en qué versión se encuentra actualmente y que características engloba. A continuación, se revisarán los distintos controladores disponibles. Algunos ejemplos de controladores que podemos encontrar son, Floodlight [11], OpenDaylight [12], ONOS [13], NOX/POX [14] y RYU [15] entre otros. Todos los controladores anteriormente enumerados son *open source*, pero también existen algunos controladores propietarios como pueden ser, Nuage VSC [16], perteneciente a Nokia, VortiQa [17] que pertenece a Freescale Semiconductor y UniFi [18] de Ubiquiti entre otros. Para finalizar se dotará de una visión general del estado de la monitorización para las redes definidas por software.

En la figura 2-1 se puede observar un esquema general en el que aparecen los elementos que vamos a utilizar, estudiar y desarrollar a lo largo del proyecto. Dichos elementos son, controlador de la red, el *switch* el cual se comunica con el controlador mediante *OpenFlow*, protocolo del cual se profundizará en breve. Otra parte importante del estudio son los hosts, ya que cada uno de ellos tiene marcada una función dentro del escenario, uno ejecutará el monitorizador/*sniffer*, otro será el encargado de transmitir el tráfico a través de la red para, posteriormente, gracias a las reglas configuradas en el *switch*, ser capturado y posteriormente procesado.

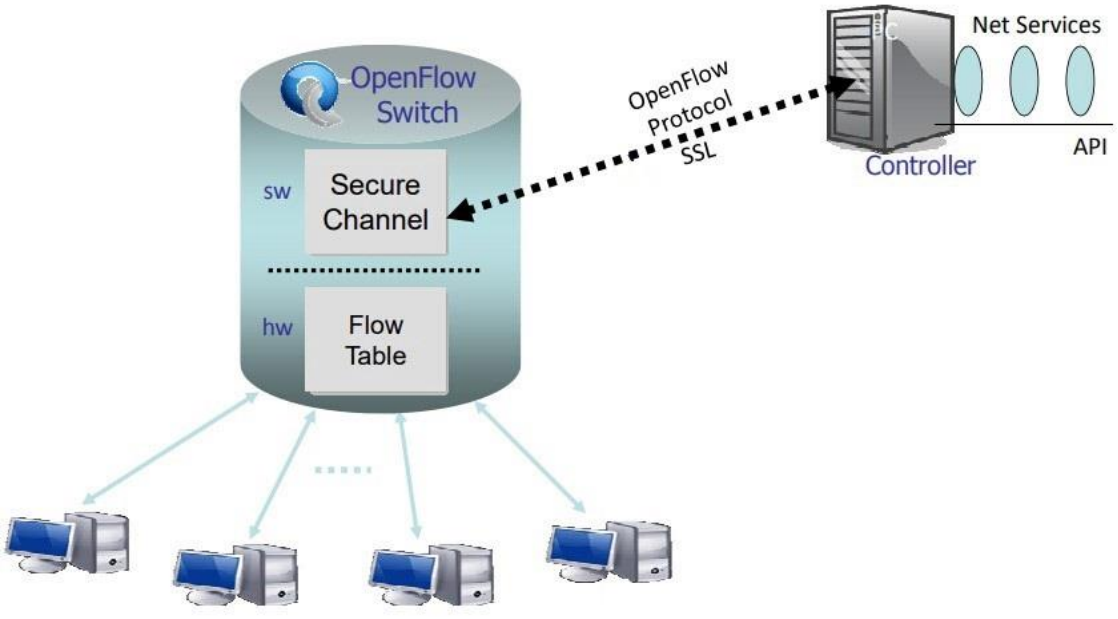


Figura 2-1: Ejemplo de elementos involucrados en una red SDN

A la hora de abordar el repaso del estado actual de los distintos componentes, se va a seguir un orden descendente. El primer paso es definir las redes SDN, redes sobre las que está fundamentado el estudio, en siguiente lugar, los distintos protocolos presentes en SDN, para seguir con la inteligencia de la red (controladores), y siguiendo el flujo, en último lugar el estado actual de la monitorización para las redes definidas por software.

2.2 Fundamentos SDN

En esta sección se va a tratar de manera general como está el estado actual de las redes SDN, así como sus principales características y el proceso evolutivo que han ido sufriendo a lo largo de los últimos años. Como comentario inicial, es una tecnología emergente la que aún necesita madurez para poder ser implantada en entornos de producción, aunque ya hay empresas pioneras que están apostando por ello.

La idea sobre la que se fundamentan las redes SDN, es desacoplar el plano de datos y el plano de control y de esta manera permitir una gestión flexible y eficiente, además de que la operación de la red se realice mediante programas software y automatismos. Trata de realizar las redes programables para ejecutar de manera más eficiente grandes despliegues y evitar errores humanos.

Se van a presentar brevemente las características que diferencian las redes SDN de las redes convencionales. En el aspecto de configuración, SDN, permite la automatización de la configuración seguida de una validación central por medio del controlador, mientras que las redes convencionales son más propensas a errores debido a las configuraciones manuales e individualizadas. En cuanto al rendimiento, SDN, aporta un control global dinámico con información entre capas de red, de la que poder nutrirse, por otro lado, las redes convencionales cuentan con información más limitada de la que valerse y configuraciones más estáticas que tardan más en ajustarse a las necesidades de crecimiento, por ejemplo. Por último, una comparación clamorosa, en el plano de la innovación observamos como SDN está mucho más abierto a la integración de nuevo software con nuevas ideas, entornos de pruebas con aislamiento de la red de producción (todo visto desde un aspecto lógico y no físico) y gran capacidad de realizar *upgrades* de manera masiva, mientras que las redes convencionales, presentan más dificultades para la introducción de nuevas ideas debido a las implementaciones hardware, también tienen entornos más limitados para pruebas y largos procesos de estabilización y estandarización.

El enfoque de la arquitectura de SDN está basada en la división entre *southbound* y *northbound.* Este enfoque aporta a la red dinamismo, gestión y adaptabilidad que encaja perfectamente con las tecnologías actuales y el rápido y constante cambio que tiene que soportar. En cuanto a los conceptos de la arquitectura SDN, *southbound* hace referencia a los elementos de red que dependen del controlador y están, “por debajo” de él. Las APIs de esta área permiten el control de la red y la realización de cambios de manera dinámica, según sea necesario. Los elementos afectados pueden ser los *switches,* por ejemplo. El concepto *northbound* hace referencia a la comunicación, “por encima”, del controlador. Es decir, la interacción con otro tipo de servicios o aplicaciones. Las aplicaciones en esta parte son como la que se ha desarrollado en este trabajo, donde pretende suplir carencias, ampliar el alcance e introducir nuevas funcionalidades.

A continuación, se muestra una figura en la que se pretende ilustrar los conceptos anteriormente mencionados sobre *northbound* y *southbound*. Cabe recalcar que estos conceptos no son exclusivos de OpenFlow y redes SDN:

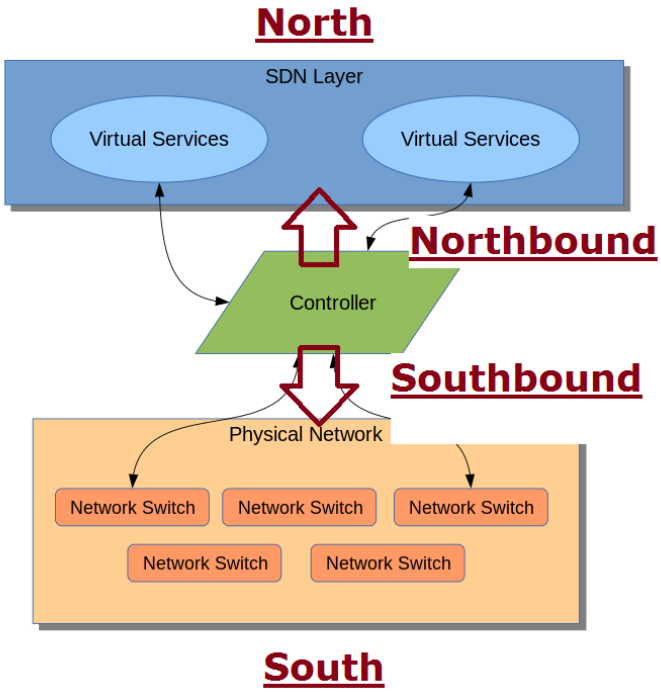


Figura 2-2: Arquitectura *northbound* y *southbound* extraído de [19]

2.3 Protocolos de SDN

Ya se cuenta con una primera noción sobre en que se fundamenta las redes SDN, se va a proseguir con el estudio de los distintos mecanismos que se pueden utilizar para comunicarse con el resto de los elementos de la red y así poder sacar partido a todas las ventajas que nos puede aportar SDN. Los principales protocolos *open source* que implementan SDN son los mostrados a continuación.

2.3.1 *OpenFlow*

El primero de los que se van a mostrar es *OpenFlow*, ya que fue el precursor y uno de los primeros estándares de SDN. Fue desarrollado en la Universidad de Stanford en el año 2008. Posteriormente la gestión y desarrollo del mismo ha pasado a manos de la ONF [20] En sus inicios, permitía la interacción entre el controlador SDN (plano de control), con los distintos elementos distribuidos por la red, como bien pueden ser *switches* o *routers* (plano de datos). También soportaba elementos físicos o virtuales en sus primeras fases.

Para poder trabajar con *OpenFlow*, es obligatorio que los dispositivos desplegados en la red soporten *OpenFlow*. Sí este requisito se cumple, los administradores de red pueden gestionar la red y aplicar políticas sobre el tráfico, así como también establecer criterios de control y reencaminamiento para mejorar el rendimiento de la red y poder probar nuevas configuraciones y funcionalidades.

En un *switch OpenFlow*, se pueden manipular las tablas de flujos en función de las necesidades de la red o de las políticas que quieren introducir los gestores de red. En la siguiente figura (2 - 3) se muestra esquemáticamente las entradas que se pueden manipular y en base a ellas tomar las acciones deseadas. Durante este proceso se van almacenando estadísticas de número de paquetes y Bytes que pueden ser de interés:

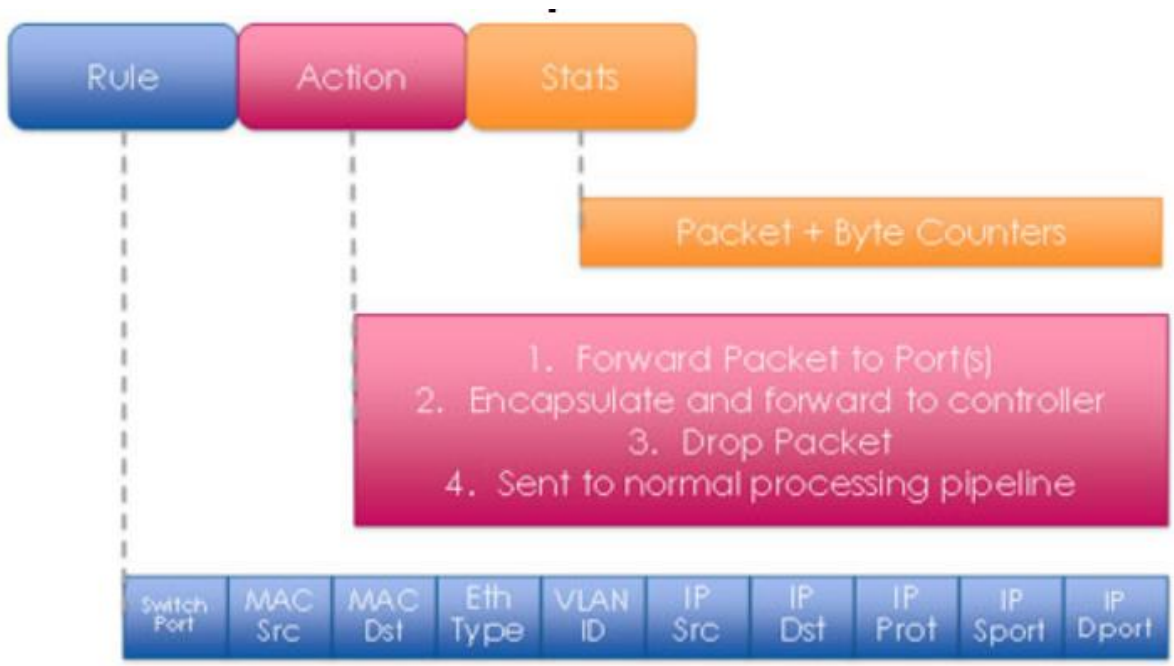


Figura 2-3: Entradas de la tabla de flujos en *OpenFlow* [21]

Las principales ventajas que ofrece *OpenFlow* son las siguientes:

- Programabilidad de la red, permite la introducción de nuevas características y servicios de una manera rápido y sencilla. Así como *upgrades* masivos.

- Inteligencia centralizada, está característica simplifica el aprovisionamiento, optimiza el rendimiento y ofrece la posibilidad de una gestión granular de las políticas.

- Abstracción de la red, este atributo hace referencia al concepto de SDN, ya que permite el desacoplamiento de hardware y software, el plano de control y el plano de datos y las configuraciones físicas y lógicas

- Flexibilidad, en términos de cómo puede ser gestionada la red ya que el software de gestión puede ser escrito usando herramientas de software al alcance de particulares o software más especializado utilizado por empresas y grandes operadores.

En la siguiente figura, 2–4, se pretende representar gráficamente el plano de actuación del protocolo *OpenFlow*:

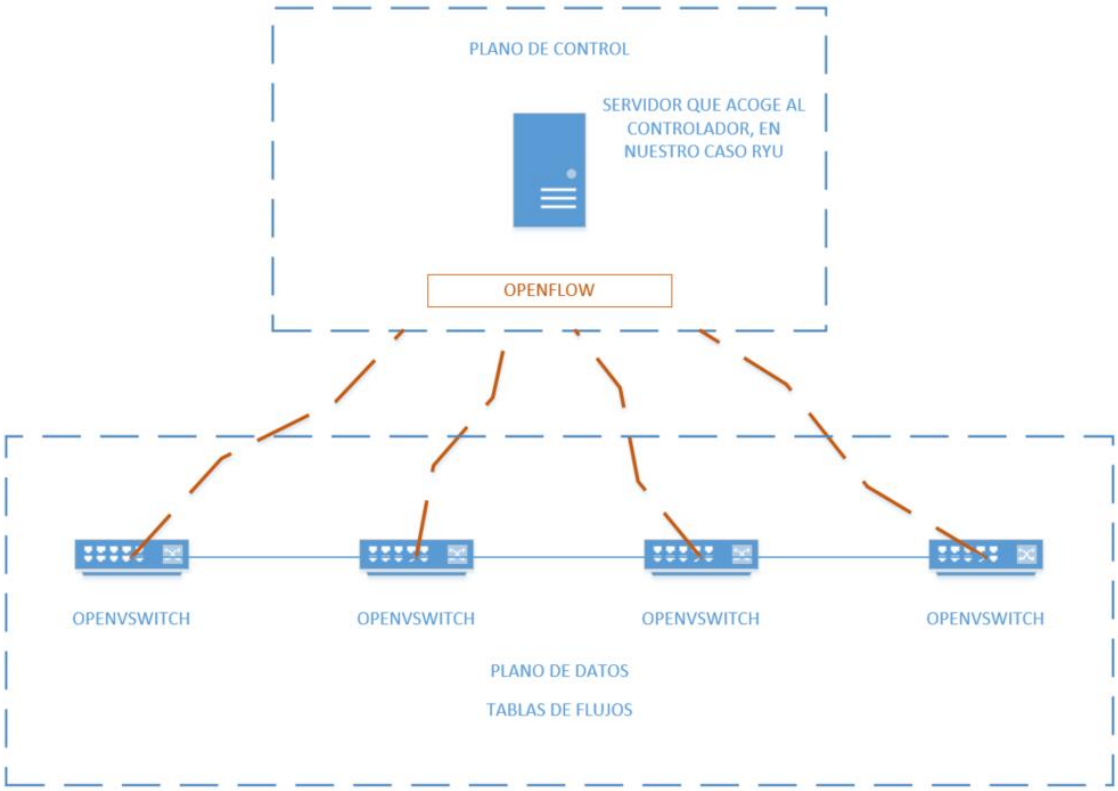


Figura 2-4: Topología clásica *OpenFlow*

Cómo último apunte cabe destacar las diferencias entre los *switches OpenFlow* y los *switches* convencionales. Las principales diferencias son que, en los *switches* convencionales el plano de datos y el plano de control (*routing*, por ejemplo) ocurren en el mismo dispositivo, mientras que en los *switches OpenFlow* estos dos planos están desacoplados, dónde el plano de datos es gestionado desde el propio *switch*, mientras que el plano de control se encuentra en el controlador de la como muestra la figura 2-4. El controlador se comunica con los *switches* mediante el protocolo *OpenFlow*.

2.3.2 **NETCONF**

El protocolo NETCONF (RFC6241 [22]), se define como aquel mecanismo que permite gestionar equipos de la red. Es decir, ofrece la posibilidad de gestionar y configurar equipos de manera unificada. Surgió para suplir las necesidades de las que otros protocolos de gestión no cubrían, cómo es la de aplicar cambios en la configuración de forma masiva en los equipos de la red.

Es un protocolo RPC (Remote Procedure Call), es decir permite desde un programa alojado en un ordenador o servidor, ejecutar configuraciones o acciones sobre una máquina remota sin preocuparse por las comunicaciones entre ambas. El lenguaje sobre el que se sustenta para interactuar con el resto de equipos remotos es XML. Las conexiones de NETCONF deben aportar autenticación, integridad en los datos, y confidencialidad. Es por

ello que en el ámbito del transporte hace uso de TLS o SSH, lo que es un punto a su favor en cuanto a seguridad del protocolo se refiere.

Es un protocolo que está teniendo gran acogida por varios fabricantes en el mercado de las redes y ya lo han integrado en sus soluciones.

La principal ventaja de NETCONF con respecto a otras herramientas similares, como puede ser SNMP es, que permite la configuración de los equipos, no únicamente la monitorización, además de que el protocolo no hace un uso excesivo de la red.

2.3.3 **RESTCONF**

Según la RFC8040 [23], en la que se describe el protocolo RESTCONF, se indica que está basado en HTTP y proporciona una interfaz programática para poder acceder a los datos mediante YANG. RESTCONF está basado en el protocolo anteriormente descrito, NETCONF, ya que hace uso de unos *data stores* y, RESTCONF, implementa un conjunto de comandos para poder acceder a ellos. La funcionalidad principal, al igual que en el caso anterior, es la configuración de equipos de manera remota y centralizada.

RESTCONF no se creó con la intención de reemplazar a NETCONF, si no que aporta una interfaz simplificada adicional que sigue los principios de tipo REST y es compatible con una abstracción de los dispositivos orientada a recursos. También RESTCONF ofrece nuevas codificaciones que NETCONF no soporta, actualmente JSON y XML.

En resumen, RESTCONF, surgió como una mejora de NETCONF ampliando funcionalidades y facilitando el manejo del mismo.

2.3.4 **OF-CONFIG**

OF-CONFIG [24], es la abreviación de *OpenFlow* Configuration and Management Protocol y se caracteriza por ser un conjunto de reglas y mecanismos para permitir a los controladores *OpenFlow* acceder y modificar las configuraciones en *switches OpenFlow*.

Una de las maneras más habituales para implementar una red definida por software, es desacoplar el plano de control de una red física y aglutinar la administración en un controlador centralizado. Dicho controlador, por norma general utiliza como protocolo para distribuir la información entre los nodos de la red *OpenFlow*. Aunque *OpenFlow* determina cómo se reenvían los paquetes entre orígenes y destinos individuales, no proporciona las funciones de configuración y administración necesarias para asignar puertos o, por ejemplo, asignar direcciones IP. Es por ello la importancia de estos tipos de protocolos, ya que ayudan en términos de las configuraciones descritas anteriormente y brindan a los ingenieros de la red una visión general de la red.

En la siguiente figura extraída de [24], se muestra gráficamente la diferencia entre OF-CONFIG y *OpenFlow*:

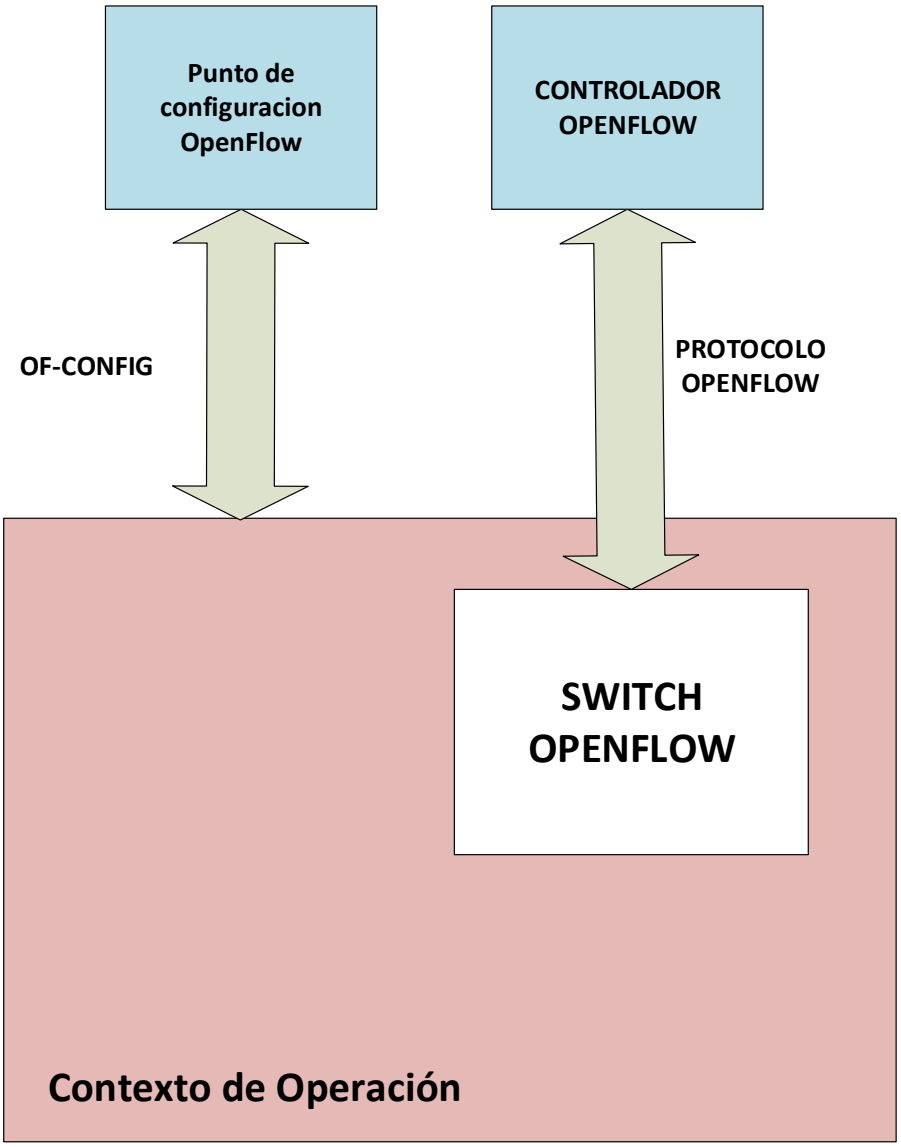


Figura 2-5: Diferencia entre OF-CONFIG y *OpenFlow* figura realizada basada en [24]

Una de las principales diferencias, a parte de la que se ha comentado anteriormente sobre la configuración de puertos e IPs, es que OF-CONFIG opera en una escala de tiempos mayor que la versión clásica de *OpenFlow* que opera a nivel temporal de flujos. La diferencia se puede ver como la necesidad de actualizar o construir las tablas de reenvíos o en contraposición habilitar / deshabilitar un puerto y no necesita hacerse en la misma escala que la actualización de los flujos, si no que se puede permitir mayor tiempo de respuesta en la configuración.

2.3.5 **OVSDB**

OVSDB hace referencia a Open vSwitch Database, el cual es un protocolo de redes definidas por software. Es un protocolo desarrollado por Nicira y posteriormente ha sido adquirido por VMware. En sus inicios era parte de OVS (Open vSwitch). Como sabemos la mayoría de los equipos de red permiten la configuración remota utilizando. El objetivo

de OVS era crear una interfaz del protocolo de administración acorde con el resto de competidores y programática. La solución encontrada para lograrlo es OVSDB.

Como ya hemos visto, a veces se puede pensar que *OpenFlow* se basta por si solo para gestionar la red, pero no es así. Para implementar SDN con controlador OVS, *OpenFlow* se utiliza para configuraciones a nivel de flujos y OVSDB se utiliza para configurar los propios OVS (interfaces, IPs…). Esto significa que se pueden crear, eliminar, modificar puertos e interfaces.
En la siguiente figura se muestra la separación entre OVSDB y *OpenFlow*:

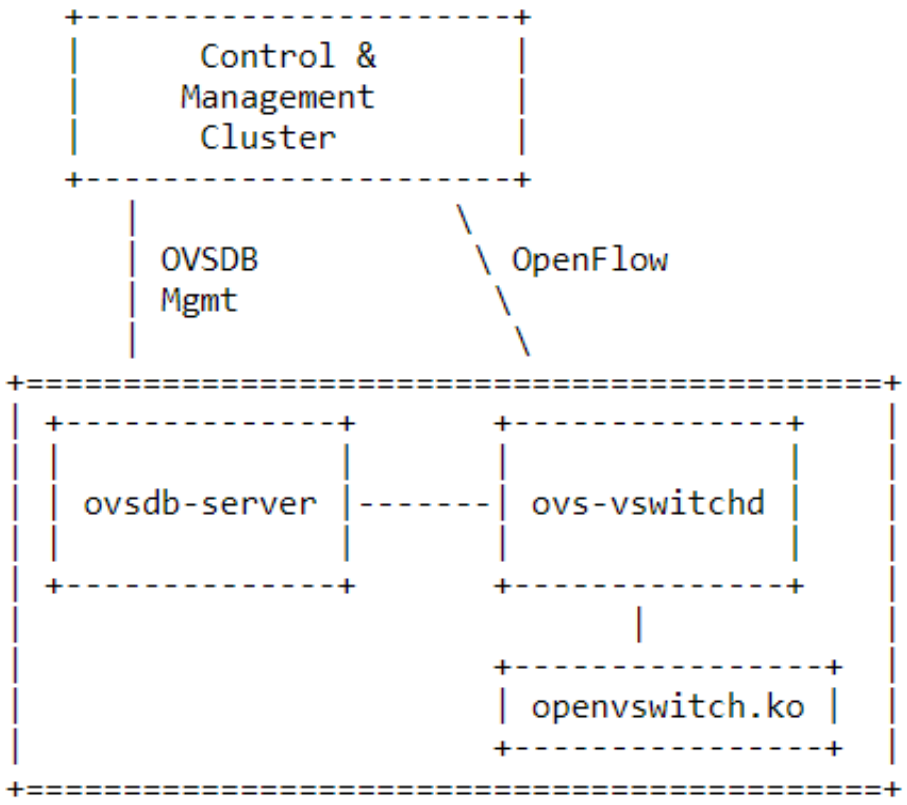


Figura 2-6: Diferencia entre OF-CONFIG y *OpenFlow* [25]

La interfaz de administración de OVSDB se usa para realizar operaciones de administración y configuración en la instancia de OVS.

Enlazando con OVSDB se va a introducir un concepto muy importante en SDN. Este concepto es VXLAN y está muy ligado a este protocolo. VXLAN se puede definir como una capa superior a la red tradicional que utiliza el protocolo IP para extender redes LAN. Para lograr esta funcionalidad se basa en el término conocido como *MAC-in-UDP*. Este término hace referencia a la agregación de la cabecera VXLAN al paquete en la capa de enlace. Este conjunto formado por la cabecera VXLAN más el paquete propio de la capa de enlace, se introduce dentro del campo de datos del datagrama UDP y se emplea IP como protocolo de la capa de red. VXLAN es una manera de crear redes SDN que ha adquirido protagonismo en los últimos años.

La relación entre VXLAN y OVSDB reside en la necesidad de utilizar un software de gestión base de datos. En estas bases de datos se guarda el etiquetado y las reglas definidas

mediante VXLAN. A continuación, se muestra la figura 2-7 que ilustra el modelo de OVSDB. Antes de ello se va a introducir el concepto de VXLAN Tunnel End Point (VTEP). VTEP son aquellas entidades encargadas de originar y terminar los túneles de VXLAN. Para la relación de VXLAN y OVSDB es necesario crear un OVSDB VTEP, este elemento se encargará de almacenar en forma de entradas en la base de datos todos los ajustes de VXLAN. El controlador se comunica con el OVSDB VTEP mediante el OVSDB:

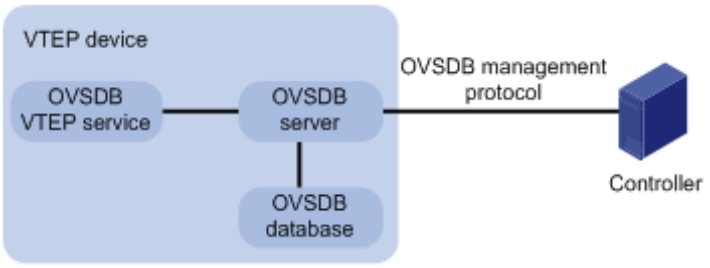


Figura 2-7:Modelo OVSDB donde se almacena ajustes y características VXLAN extraído de [26]

2.4 Controladores de redes SDN

Una vez repasado el estado global de los protocolos de SDN, se va a continuar con el estado en el que se encuentran cada uno de los elementos involucrados en el control y, algunos ejemplos típicos que podemos encontrarnos para implementar el escenario bajo estudio. En este caso, se va a aportar una breve visión sobre los controladores disponibles *open source* para SDN. Durante el breve repaso se va a comentar principalmente, el lenguaje en el que está implementado, si se puede integrar con Mininet y Open vSwitch, el nivel de documentación disponible y las versiones de *OpenFlow* que soportan, entre otros aspectos que pueden tener relevancia en función del controlador que se esté analizando. Cabe destacar que, la última versión de *OpenFlow* es la 1.6, pero es únicamente accesible para los miembros de la ONF, por tanto, la versión más moderna es la 1.5.1.

2.4.1 Floodlight

Floodlight es un controlador para SDN que usa el protocolo *OpenFlow* para gestionar y orquestar los flujos de tráfico. Las versiones soportadas a día de hoy de *OpenFlow* son desde la 1.0 hasta la 1.4 (1.5 si se dispone de Floodlight Master). Entre sus características principales se puede destacar que está implementado en Java, cuenta con REST APIs que permiten que sea más sencillo integrar el controlador con distintas soluciones. Es un controlador muy modular lo que favorece se adaptación a diversos tipos de entornos.

En cuanto a virtualización, soporta Open vSwitch y Mininet. Es válido para el uso en equipos físicos que hablen *OpenFlow*. Un aspecto importante a la hora de seleccionar el controlador para un despliegue o para realizar un estudio, es la cantidad y calidad de la documentación disponible para poder realizar consultas durante el desarrollo. Este controlador presenta buen nivel de documentación. No está soportado en OpenStack.

2.4.2 **OpenDaylight**

OpenDaylight es muy similar en cuanto a la implementación con respecto al anterior controlador. En este caso, se encuentra dentro de una Java Virtual Machine (JVM). Esto significa que se puede implementar sobre hardware o en cualquier plataforma que soporte Java. Es un controlador modular, lo que permite a los usuarios adecuar el controlador en función de las necesidades de la red. Soporta varios tipos de protocolos de SDN, entre ellos está *OpenFlow* con versiones soportadas de la 1.0 a la 1.4, y también soporta NETCONF, OVSDB, PCEP entre otros.

En el ámbito de la virtualización soporta Open vSwitch y Mininet. En cuanto al nivel de documentación se situaría en un punto bastante ventajoso, puesto a la gran variedad de foros y guías de despliegue del controlador. Soportado en OpenStack.

2.4.3 **NOX/POX**

NOX y POX son dos controladores SDN. Han sido agrupados bajo el mismo apartado ya que POX es la versión de NOX sólo que implementado en Python. NOX está implementado en C++. A diferencia de los dos controladores anteriores presenta un nivel bajo de modularidad factor muy importante a tener en cuenta. Ambos sólo soportan la versión 1.0 de *OpenFlow*.

Soporta la virtualización de *switches* mediante Open vSwitch y Mininet. Y en el ámbito de la documentación se queda en una posición más atrasada que con los controladores anteriores, aunque cada vez está ganando más protagonismo y se pueden encontrar diversos estudios sobre estos controladores. No está soportado para OpenStack.

2.4.4 **RYU**

El RYU es el controlador seleccionado para el estudio que abarca este trabajo de Fin de Máster. Está desarrollado en Python, presenta un nivel aceptable de modularidad. Actualmente cuenta con módulos de monitorización, entre otros. Es compatible con varios protocolos SDN, entre los que destacan NETCONF, OF-CONFIG y *OpenFlow* con las versiones de la 1.0 hasta la 1.5 añadiendo las extensiones de Nicira. Nicira era una empresa que se dedicaba a SDN y a la virtualización de redes que fue comprada por VMware y es una parta activa de VMware para el desarrollo en estos ámbitos.

Es compatible con Mininet y con Open vSwitch, que son las herramientas en las que nos hemos apoyado para poder virtualizar el escenario y poder realizar pruebas. En el ámbito de la documentación es el que más visión tenemos y se puede decir que está muy documentado de manera global, con numerosas guías. Sí está soportado en OpenStack.

2.4.5 **Cherry**

Se ha añadido el controlador SDN Cherry por las curiosidades que presenta y es un claro ejemplo de controlador dedicado para un uso en concreto. Entre las principales curiosidades se aprecia que, el controlador, está implementado en lenguaje Go, el cual es un lenguaje basado en programación concurrente y está inspirado en C. Dicho lenguaje ha

sido desarrollado por Google. También, se puede integrar con switches comerciales que ejecuten *OpenFlow*. Soporta las versiones de *OpenFlow* de la 1.0 a la 1.3.

Es compatible con Open vSwitch y con Mininet. En el ámbito de la documentación para el desarrollo es bastante pobre. No presenta soporte para OpenStack. Este controlador está orientado para SDN en proveedores de internet, lo que no encaja con el tema tratada en este trabajo.

2.4.6 **Trema**

El controlador Trema está desarrollado en Ruby y C. La modularidad que presenta se puede catalogar como intermedia, ya que sí que permite dicha característica y ofrece sus propios módulos. Soporta la versión 1.3 de *OpenFlow* a través de un repositorio llamado *TremaEdge* [27]. Trema no ofrece un controlador NETCONF que permita el soporte de OF-CONFIG.

Se puede integrar con Mininet y con *switches* Open vSwitch. En cuanto al nivel de documentación sobre la herramienta se puede afirmar que está bien documentada y se pueden encontrar ejemplos e implementaciones en los repositorios que ofrecen los desarrolladores. No soportado en OpenStack.

2.4.7 **ONOS**

El controlador ONOS (Open Network Operating System) está implementado en Java y ofrece una interfaz de usuario web. Entre sus virtudes principales se aprecian escalabilidad, modularidad, alto rendimiento en las redes y fiabilidad, entre otros. Implementa hasta la versión 1.5 de *OpenFlow*.

Ofrece un alto volumen de documentación, en la que se pueden encontrar, por ejemplo, guías de integración con Mininet y *switches* Open vSwitch. Si soporta integración con OpenStack.

2.4.8 **Tabla resumen controladores SDN**

A continuación, se muestra una tabla resumen de las características de las controladoras SDN *open source* analizados:

	Lenguaje	Integración Mininet	Integración OpenvSwitch	Nivel Documentación	Versiones OpenFlow	Modularidad	Openstack
Floodlight	Java	Sí	Sí	Buena	1.0 - 1.4	Media	No
OpenDayLight	Java	Sí	Sí	Buena	1.0 - 1.4	Alta	Sí
NOX/POX	C++/Python	Sí	Sí	Moderada	1.0	Baja	No
RYU	Python	Sí	Sí	Muy buena	1.0 - 1.5	Media	Sí
Cherry	Go	Sí	Sí	Pobre	1.0 - 1.3	Baja	No
Trema	Ruby/C	Sí	Sí	Moderada	1.3	Media	No
ONOS	Java	Sí	Sí	Buena	1.5	Alta	Sí

Tabla 2-1: Tabla comparativa características básicas de controladores SDN

2.5 Monitorización Redes SDN

Una vez que se cuenta con una visión general sobre los fundamentos, las ventajas de SDN y los elementos que forman parte de la red, se puede abordar el tema que pretende abarcar este trabajo de fin de máster, el estado actual de la monitorización de SDN utilizando *OpenFlow*.

Para recoger el estado de la red, los controladores pueden obtener y acumular información de interés y de esta manera poder construir una visión global de la red y aportar a la capa de aplicación con información útil, por ejemplo, la topología de la red para tomar decisiones sobre ella. Los flujos que atraviesan la red aportan información sobre, tiempos de duración, número de paquetes, tamaño y ancho de banda. De manera general, la adquisición de todos estos datos se hace de la siguiente manera; cada elemento de *switching* recoge y almacena en local las estadísticas sobre el tráfico. Estas estadísticas almacenadas localmente por los *switches* pueden ser recuperadas por los controladores (modo "*pull*"), o proactivamente puede ser reportado a los controladores (modo "*push*").

Un método común para manejar los datos recogidos es mediante las denominadas *Traffic Matrix (TM).* Las TM reflejan el flujo que circula entre los posibles pares, origen-destino, de la red. Existen varias maneras de manejar la información extraída de la red, TM es una de ellas, pero también podemos encontrar OpenSketch [28].

Un concepto indispensable para comprender SDN con *OpenFlow*, son las tablas de flujos, tablas generadas en los *switches* a medida que el tráfico va atravesando la red. Los *switches* realizan una búsqueda en los paquetes y un reencaminamiento de los paquetes en función de las distintas entradas de la tabla de flujos. Mediante *OpenFlow*, el controlador es capaz de añadir, actualizar y borrar las tablas de flujos en base de las necesidades. Cada tabla de flujos contiene entradas, cada entrada consiste en *match fields*, contadores y conjunto de

instrucciones que aplicar a los paquetes que concuerden. Podemos hacernos una idea de la utilidad y el alcance que esto aporta a las redes y la gran cantidad de configuraciones a la hora de tomar decisiones en función a escenarios. Para aportar aún más capacidad de configuración se pueden asignar prioridades a las entradas de la tabla de flujos para que se evalúen antes que otras en función de las necesidades que se requieran.

A continuación, se muestra en la figura 2-8 el camino, a alto nivel, que recorre un paquete cuando atraviesa un *switch* con *OpenFlow*. Se puede observar cómo puede ir pasando por varias tablas en función de las concordancias que se vayan produciendo y en función de ellas también existen unas acciones que se pueden definir para, en último lugar, salir a la red al destino que corresponda.

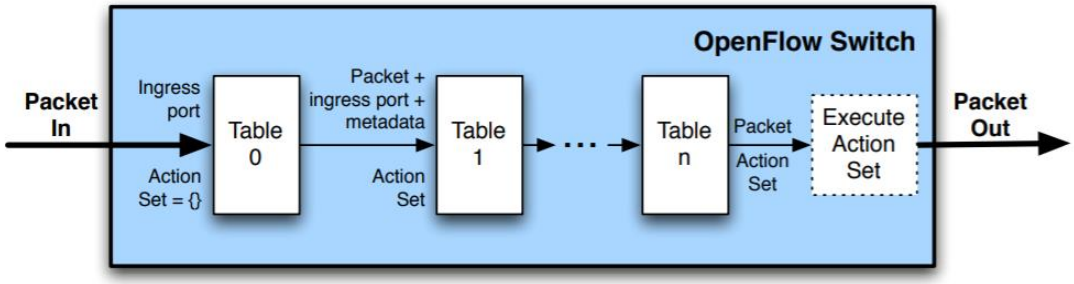


Figura 2-8: Flujo de un paquete a través de *switch OpenFlow* [29]

Retomando el tema que va a abarcar este Trabajo de Fin de Máster sobre la monitorización, se va a hacer un breve repaso sobre los principales retos y los métodos que se están llevando a cabo para tal propósito. La necesidad de monitorizar las redes utilizando *OpenFlow* surge por el gran volumen y la diversidad de tipos de datos que atraviesan la red y cada uno de ellos necesita un trato distinto para cumplir su propósito. Las redes actuales presentan los siguientes problemas que pueden ser abordados mediante la monitorización de la red. El tráfico actual hace difícil a los operadores de red medir, en pequeñas escalas de tiempo, el estado y el dinamismo de la red de una manera efectiva. También es complicado y tedioso realizar ingeniería de tráfico con las redes convencionales para garantizar el buen rendimiento de las aplicaciones y, en caso que sea necesario, detección de congestiones en la red. Otro punto a tener en cuenta es la consecución de una QoS de la red para satisfacer las necesidades de los usuarios.

Los *routers* y *switches* convencionales, son inflexibles y no pueden manejar de la manera más apropiada, en muchos de los casos, los diferentes tipos de tráfico, debido a la manera en la que están implementadas las reglas de *routing*. *OpenFlow* permite solventar este tipo de problemas.

Anteriormente hemos introducido los dos métodos mediante los cuales el controlador puede obtener información de la red, para a partir de ellos, tomar decisiones y comprobar la salud de la red, y en caso que sea necesario, anticiparse a problemas futuros. Existe el método pasivo y el método activo. Mediante el método activo los flujos de la red son constantemente monitorizados para comprobar el rendimiento mediante envío de paquetes de muestra por los distintos caminos que forman la red. El propósito de este método es, por ejemplo, medir el RTT, o ajustar las políticas de *forwarding* en función de las necesidades, una de las principales desventajas de este método es que introduce tráfico en la red que en momentos de picos de carga puede ser contraproducente. En cambio, en las medidas pasivas, existen una serie de sondas lo largo de la red donde el tráfico real va

pasando y se va capturando y analizando. En nuestro estudio nos vamos a basar en este segundo método de monitorización en el que vamos a analizar tráfico real mediante replicar el tráfico por un puerto del *switch* donde se encuentra el monitorizador/*sniffer*. Las medidas pasivas no son intrusivas ya que no inyectan ningún tipo de tráfico a la red.

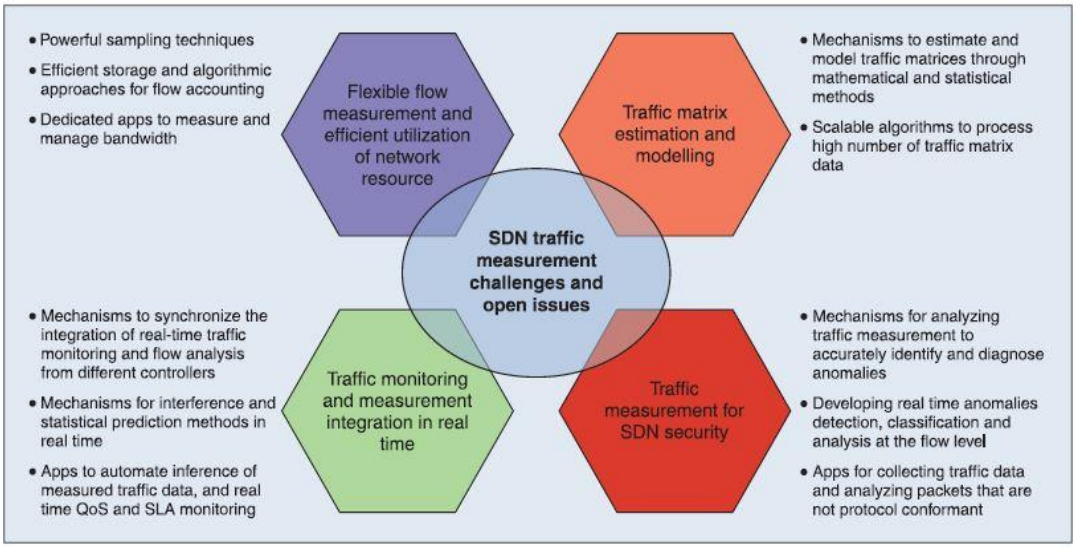


Figura 2-9: Retos y casos abiertos de la monitorización en SDN [30]

En la actualidad y a medida que SDN y *OpenFlow* siguen mejorando se pueden definir tres corrientes en la monitorización El primer camino se centra en encontrar el denominado *"balance in overhead implications"* usando técnicas de muestreo de tráfico, agregación, búsquedas inteligentes, etc. La segunda corriente presta más atención al compromiso de uso de los recursos disponibles mediante medidas de precisión. En esta segunda corriente se puede monitorizar el estado de CPU de los switches. Y en último lugar se encuentra la corriente que está principalmente centrada en dotar a la red de una medida en tiempo real del tráfico para poder tomar decisiones tanto de manera reactiva, cómo proactiva. Este Trabajo de Fin de Máster está enfocado hacía la tercera corriente.

Se va hacer un breve repaso comparativo de los métodos de monitorizar el tráfico en SDN. En primer lugar, se va a comenzar por el método descrito en [30]. Es un método activo, el mecanismo consiste en medir agregación de tráfico a gran escala usando las reglas de *matcheo* de los *switches*. Resumidamente, el resultado del proceso es que reduce la carga, pero requiere de una constante actualización de las reglas de los *switches*. En siguiente lugar, OpenNetMon [31], también es un método activo y se basa en una recuperación de datos adaptativa desde los *switches*. Ofrece una precisión elevada a medida que se produce sobrecarga en la red. iStamp [32], es un método de monitorización activo y se basa en las particiones TCAM para agregar y desagregar el tráfico. Mejora la precisión de las medidas de los dos métodos anteriores, pero necesita de un mecanismo adicional para marcar los flujos importantes y hacer el seguimiento y recoger estadísticas. Otro método interesante es el descrito en [33], el cual es un método activo, dónde se instruyen *switches* basados en hash, para recoger la información del tráfico. Tiene alta precisión, pero las reglas de monitorización deben ser cuidadosamente delegadas por la red. En [34] se presenta un algoritmo basado en la predicción de errores para detección de anomalías en la red. Es un método activo y muy efectivo para la identificación de tráfico. OpenTM [35], es un método activo que está constantemente sondeando los *switches* para obtener estadísticas, alta

precisión y también alta carga para la red. Payless [36] se caracteriza por ser un método activo, y utiliza un método adaptativo para el sondeo de las estadísticas de los flujos con un alto o bajo intervalo de frecuencia. La precisión y la sobrecarga varían en función del tamaño del intervalo de sondeo. PLANCK [37], utiliza el *port mirroring* que está disponible en la mayoría de *switches* convencionales, es un método activo, es rápido para obtener estadísticas, pero el principal problema es que el tráfico puede exceder la capacidad de los puertos. En siguiente lugar encontramos OpenSample [38], es uno de los pocos métodos pasivos que encontramos en el panorama actual de monitorizadores. Su funcionamiento se basa en sFlow. SFlow es la abreviatura de "*sample flow*", que es un estándar para la exportación de paquetes de la capa 2. Presenta una serie de estadísticas que permiten el monitoreo de la red. Tiene gran precisión, baja latencia y no es intrusivo en la red ya que es un método pasivo. Requiere de sondas desplegadas por la red para adquirir las medidas. Para finalizar con los monitorizadores seleccionados, ya que existen más, y algunos que están en desarrollo aún, encontramos OpenSketch, es un método de monitorización activo, usa una serie de librerías en el plano de control de la red que configuran automáticamente y gestionan los recursos para las actividades de monitorización. Este método de monitorización ofrece baja latencia y alta precisión en las medidas.

Una vez repasado todo el panorama actual, se va ubicar nuestra aproximación de monitorización en comparación con los métodos anteriormente comentados. El sistema implementado se basa en un método pasivo de monitorización, ya que no requiere de inyección de tráfico especifico durante el proceso. Se vale del tráfico que atraviesa la red. Para ello está basado en un *port mirroring* en el *switch* como se ha mencionado en PLANCK. El *switch* es quién replica todo el tráfico por la interfaz deseada. El tráfico va a parar al *sniffer/*monitorizador donde se extraen los datos de interés para luego ser representados y en caso que sea necesario tomar decisiones.

Tras realizar el breve repaso sobre las herramientas de monitorización actuales que están disponibles en el ambiente científico, podemos situarnos y dirigir nuestra investigación para alcanzar objetivos de interés para la gestión de las redes modernas. Como hemos visto, la mayoría de las herramientas y métodos de monitorización se centran en el nivel 2 de la capa OSI, nuestro estudio va a ir por encima de ello, en la capa de aplicación. Una combinación de las técnicas que se ha desarrollado con alguna de las mencionadas anteriormente puede aportar capacidad de configuración, anticipación a errores, ajuste dinámico a las necesidades de la red en todo momento y de manera automática y rápida sin tener que esperar a las configuraciones manuales que requieren más tiempo y son propensas a errores.

2.6 Conclusiones

En este apartado se ha introducido y repasado las principales partes involucradas, en cuanto a controladores y protocolos SDN se refiere, así como sus principales características, ventajas y desventajas. Es importante tener esta visión general para entender el motivo de la elección de los distintos elementos y así poder lograr alcanzar el objetivo final del trabajo de Fin de Máster.

En el próximo capítulo se detalla cómo se ha implementado el escenario y el motivo de la elección de los elementos implicados.

3 Diseño

3.1 ***Introducción***

En la sección de diseño se va a desarrollar y explicar la elección de los elementos que forman parte del estudio con las justificaciones correspondientes del porqué de las decisiones tomadas y como ayudan más a nuestros intereses que, por ejemplo, otros controladores u otros elementos de emulación de la red SDN.

Como primera toma de contacto, el entorno de pruebas está formado por 3 hosts, donde cada uno tiene una función específica dentro del escenario, un switch y un controlador. Las etiquetas con las que se ha denominado cada componente son, inyector, colector, controlador, *sniffer*/monitorizador y, en último lugar, el bloque de visualización de dashboards.

El escenario a alto nivel con las funciones de cada componente se muestra a continuación en la figura 3-1:

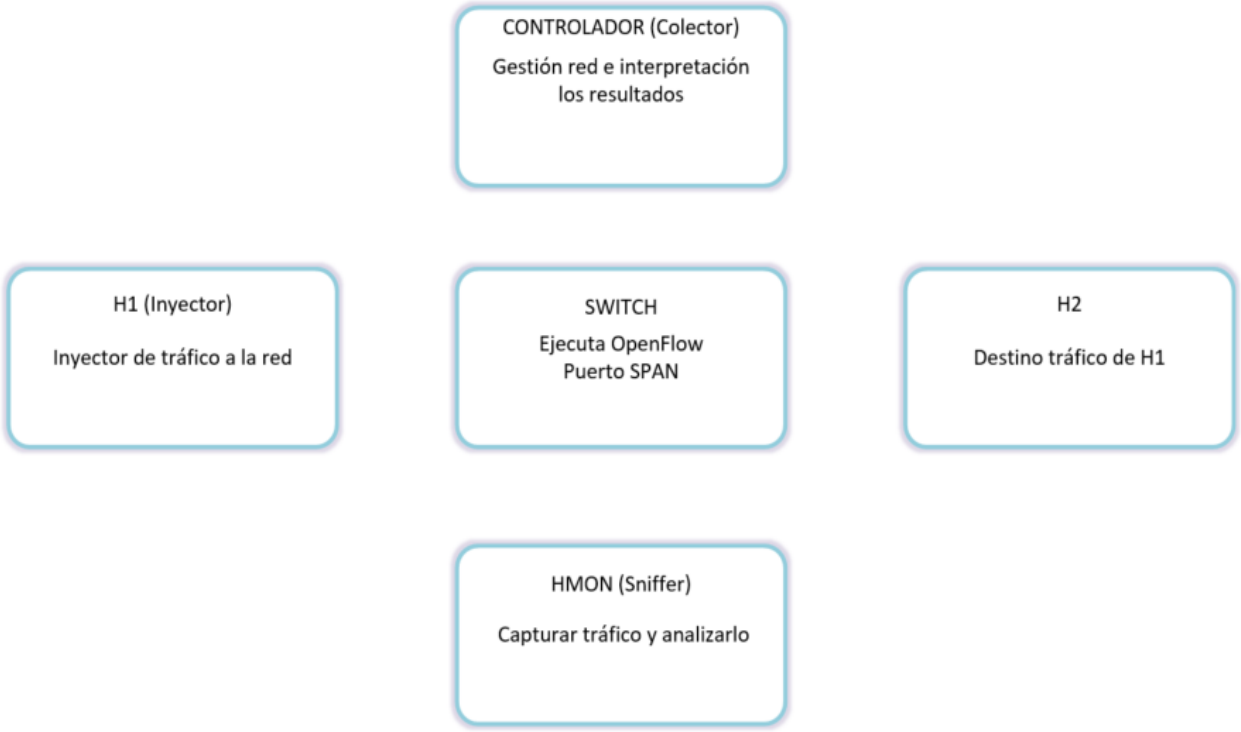


Figura 3-1: Escenario bajo estudio con funcionalidades y componentes a alto nivel

Uno de los componentes es el encargado de transmitir el tráfico por la red, este host se denomina en el estudio “H1”. El siguiente host, llamado “H2”, es el encargado de recibir el tráfico de “H1”. Y por último es el host, llamado “HMon” el que hace el papel de una sonda de monitorización en la red. El último host, “HMon”, está conectado a un *switch*, a través de un puerto de SPAN. Los componentes anteriormente mencionados hacen referencia a los extremos de la red. En la topología existe una capa de acceso que interconecta los tres hosts y se corresponde con un *switch* el cual es compatible con *OpenFlow*. En último lugar entra en juego la inteligencia de la red, el controlador, que instala reglas para monitorizar en el switch y, además, ejecuta el colector de los datos extraídos por el *sniffer* para posteriormente ser representados en un dashboard.

Cómo se puede observar en la figura 3–1, se ha dado una visión de manera esquemática la topología lógica sobra la cual se va a basar nuestro estudio, con una breve descripción de la función principal de cada componente dentro del escenario. En esta primera aproximación no se ha entrado en el flujo que siguen los paquetes durante el proceso.

3.2 ***Decisiones de diseño***

En este apartado se van a indicar las decisiones de diseño, del controlador y del programa de emulación de la red SDN

3.2.1 **Controlador RYU**

A la hora de identificar un controlador de los disponibles hay una serie de características en las que nos debemos centrar y son indispensables antes de entra en detalle de funcionalidades, que también son muy importantes.

En primer lugar, uno de los factores importantes a la hora de seleccionar un controlador para SDN es el lenguaje de programación en el que estén desarrollados. El lenguaje de programación es importante tanto en cuanto, cada lenguaje ofrece distintas características sobre permitir *multithreading*, sencillez para aprender el lenguaje o preferencia según lenguajes que ya se conocen, acceso de memoria rápido, buena gestión de la memoria y multiplataforma. Es importante tener en cuenta todos esos factores porque dependiendo del controlador que elijamos se verá afectado el rendimiento y la velocidad de desarrollo. Un aspecto muy importante en el momento de tomar la decisión de escoger el lenguaje de programación es que permita hacer prototipado rápido, ha sido crucial para la selección del controlador. Los lenguajes más usados son Python, C++ y Java.

Otro aspecto a tener en cuenta para seleccionar el controlador es si es compatible con el protocolo SDN que se va utilizar. Para elegir nuestro controlador nos aseguramos de que soporta *OpenFlow*.

Otra de las características en las que se debe hacer más énfasis, después de haber realizado el estudio, es la capacidad de programabilidad de la red. Pero, ¿por qué es tan importante? La respuesta es sencilla, SDN, una de las ventajas más grandes que ofrece respecto a las redes convencionales es la programación de las redes, y de esta menare poder gestionar de manera dinámica las redes. La creación de scripts que automaticen las funciones y decisiones de la red y la capacidad de añadir aplicaciones en la plataforma del controlador es un aspecto que puede llegar a decantar la decisión de escoger un controlador u otro.

En siguiente lugar ya entramos en aspectos de rendimiento, fiabilidad, escalabilidad y seguridad que nos pueden aportar los distintos controladores. la obtención de resultados concluyentes sobre el rendimiento de los controladores y la comparación de cada uno de ellos está fuera del ámbito de este trabajo. Por tanto, nos hemos basado en un estudio [39] en el cual hacen una comparativa de los distintos controladores del mercado.

Se han expuesto todos los parámetros en los que nos hemos basado para seleccionar el controlador para realizar el experimento. El controlador que se ha escogido ha sido RYU. La elección viene motivada porque RYU está desarrollado en Python, un lenguaje sencillo y flexible. En siguiente lugar se verificó que RYU es compatible con *OpenFlow*, y lo es. Es compatible con las versiones 1.0, 1.2, 1.3 y 1.4, pero no solo eso, además es compatible con otros protocolos que se utilizan en SDN, como son NETCONF y OF-CONFIG. Esto lleva a que se puede hacer un estudio futuro muy similar a este, pero en vez de con

OpenFlow con esos dos protocolos. RYU también se escogió gracias a la programabilidad que ofrece, a la documentación disponible y a los foros que tiene activos. En cuanto a la eficiencia no es de los mejores de los disponibles, presenta una latencia elevada y un *throughput* bajo, pero para la creación de una maqueta de monitorización nos sirve, porque además implementa y facilita la incorporación de funciones de monitorización.

3.2.2 **Entorno de pruebas**

Para poder simular la red y llevar a cabo el estudio, se ha escogido como herramienta de virtualización de redes, Mininet. Mininet permite crear una red virtual en un ordenador simulando hosts, *switches* y ejecutando un controlador de la red. El alcance de las herramientas de virtualización es enorme, ya que ofrecen la posibilidad de emular la red y realizar pruebas sobre ellas sin desplegar ningún tipo de hardware. Gracias a ello, si las pruebas previas han sido favorables y se han alcanzado los objetivos deseados se puede proseguir con la inversión y realizar el despliegue real. Esta es una bondad de realizar un despliegue virtual previo a un despliegue físico con todo el gasto que conlleva.

Para nuestro caso únicamente vamos a implementar el escenario de manera virtualizado y sobre él, ejecutar los distintos componentes involucrados bajo el estudio. El escenario escogido virtualizado en Mininet es sencillo, está formado por tres hosts, un switch, un controlador y los correspondientes enlaces de conexión.

Las principales ventajas que presenta Mininet y por la cual se ha escogido como sistema de emulación, son las siguientes. Mininet es compatible con un amplio espectro controladores y con prácticamente cualquier software desarrollado sobre GNU/Linux, si lo comparamos con NS-3 [40] esta herramienta no es compatible con controladores reales. Mininet presenta una escalabilidad media para múltiples procesos, lo que para nuestro estudio es suficiente. También, Mininet cuenta con una GUI, la cual sirve para la observación gráfica de la red y permite programar scripts de Python que configuren la red y ejecuten pruebas de forma automatizada

La red que hemos diseñado para realizar el estudio en la GUI de Mininet presenta la siguiente topología mostrada en la figura 3-2

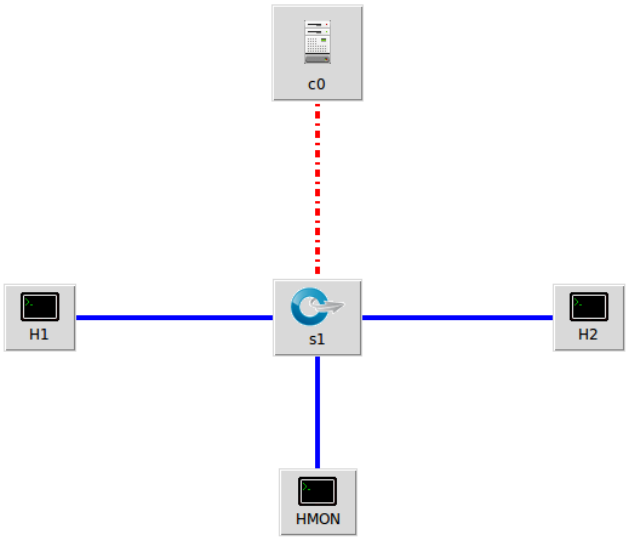


Figura 3-2: Escenario bajo estudio en GUI de Mininet

En lo que atañe a este trabajo, se ha utilizado principalmente la configuración de la red mediante CLI y scripts ya que no se ha generado una red muy compleja porque el estudio se centra en la monitorización, y con un *switch* y tres hosts es suficiente para alcanzar el objetivo. No se ha utilizado la GUI de Mininet.

3.2.3 Diseño de elementos del framework

Este subapartado, presenta el diseño de los distintos elementos existentes en el entorno de pruebas. Para presentar los distintos elementos se va seguir el recorrido de un paquete desde que se origina hasta que se representa. Para terminar con el esquema lógico con el recorrido de los flujos en global.

El *pipeline* de proceso de un paquete por la red comienza en el inyector de tráfico. Es uno de los elementos más simples de la red y no se ha realizado gran implementación en esta parte. Pero si da pie a hablar del tipo de tráfico que se ve a encargar de transmitir. Se han definido tres tipos de tráficos que se van a transmitir, concretamente HTTP, SSL/HTTPs y DNS ya que son los protocolos que entran en juego en la navegación Web

De cada tipo de tráfico interesa extraer distinta información, por ejemplo, para HTTP interesa extraer el host al que nos conectamos, de SSL también interesa el host, pero se realiza de manera diferente que en el caso anterior por las propias características del protocolo. Por último, en DNS, se desea conocer la IP que resuelve la consulta y el dominio al que se pretende conectar. La implementación para extraer toda esta información en el sniffer se detalla en el capítulo de implementación.

En siguiente lugar siguiendo el recorrido de un paquete por la red encontramos el *switch.* El *switch* tiene parte de diseño, pero por las características de SDN el comportamiento del mismo se define desde el controlador. A continuación, en la figura 3-3, se va a representar el diseño del switch *OpenFlow*:

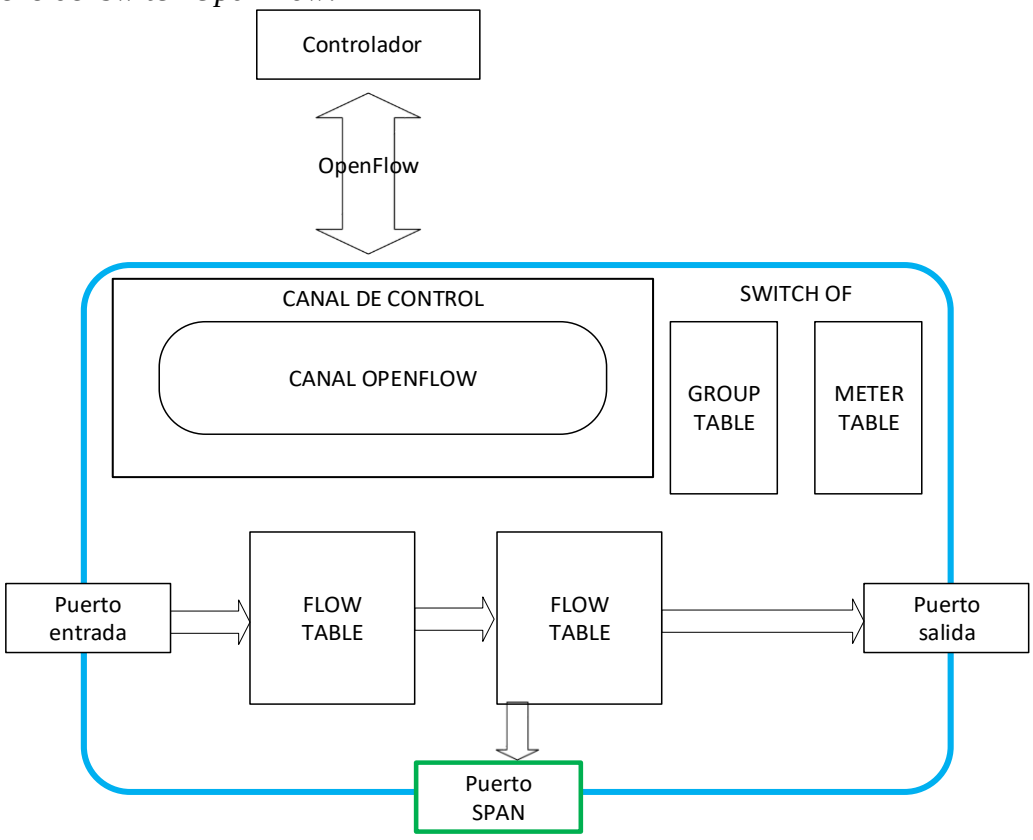


Figura 3-3: Diseño switch del escenario a alto nivel

Se ha modificado una regla para que se replique todo el tráfico por el puerto en el que se encuentra conectado el *sniffer* a parte del destino al que deba dirigirse el tráfico. También es importante indicar al *switch* la versión de *OpenFlow* que debe ejecutar. En nuestro caso se ha utilizado la versión 1.3.

Cuando los paquetes se replican por el puerto de SPAN, van a parar en el *sniffer* el esquema de diseño, a alto nivel, del *sniffer* se muestra a continuación:

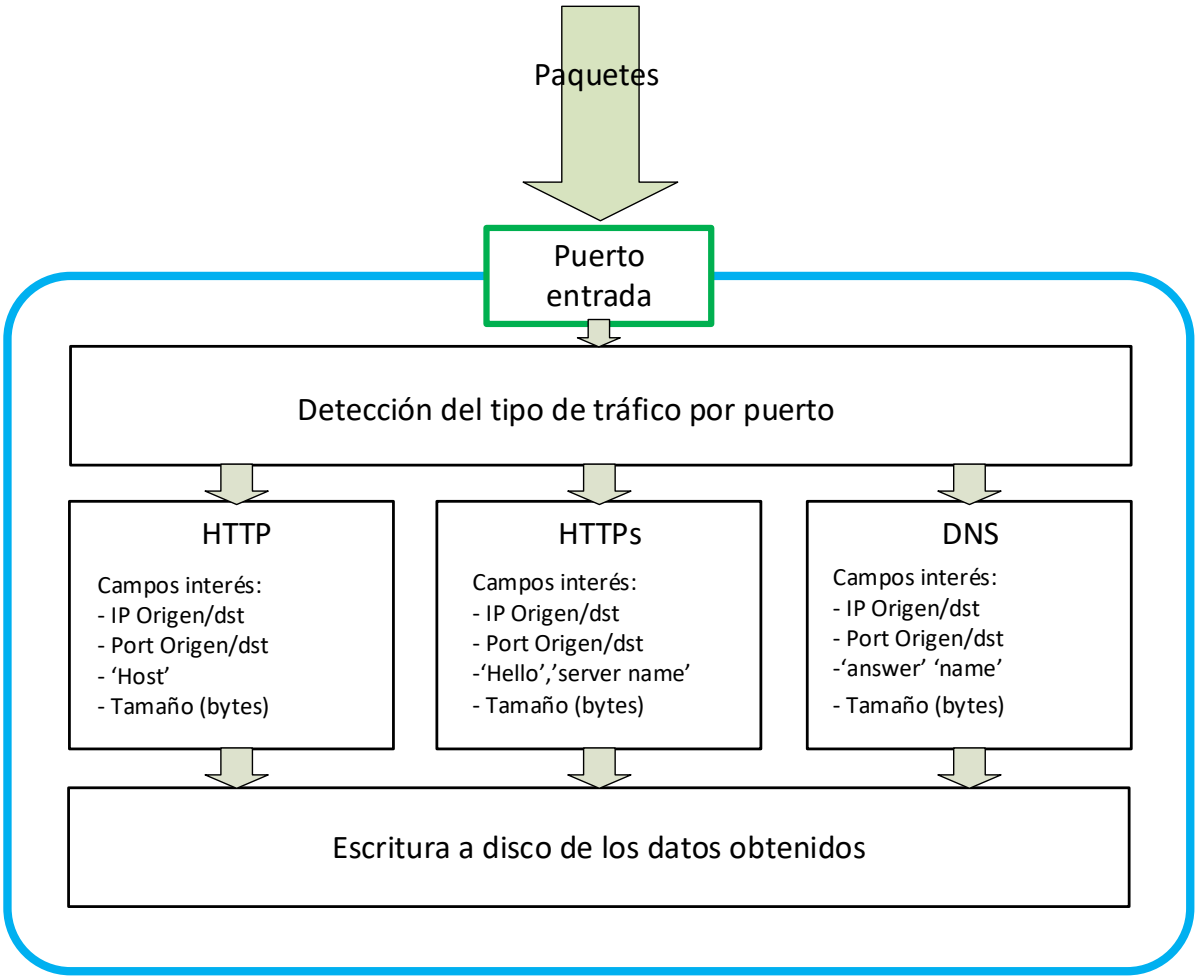


Figura 3-4: Diseño *sniffer* del escenario a alto nivel

Los criterios de diseño del *sniffer* han sido, en primer lugar, la separación por tipo de tráfico mediante un filtrado simple por puerto. Una vez que se ha diferenciado el tipo de tráfico, es necesario tratar de manera distinta cada uno. Al final, se obtienen los campos de interés y se escriben en disco los resultados con el mismo formato por cada tipo de tráfico. Se desarrollará a continuación con más detalle, los campos concretos que se almacenan.

Por cada tipo de tráfico se han seleccionado unos campos de interés. Estos campos de interés están detallados en la figura superior y son, para HTTP la IP origen y destino, puerto origen y destino, el host que resuelve la petición y el número de bytes del paquete. Para SSL/HTTPs se ha extraído la IP origen y destino, puerto origen y destino y el host que resuelve la petición. Para ello era necesario capturar el paquete de *Client Hello* característico de este SSL y extraer el campo Server Name Indication (SNI). También se recolecta el número de bytes del paquete. En último lugar para DNS, se extraen los mismos campos de IPs y puertos, pero para la extracción del dominio al que se conecta, primero es necesario comprobar que es un paquete de respuesta DNS. La IP que se almacena en el

listado del tráfico DNS es la que resuelve la consulta y no la IP de destino a quien iba dirigida la petición. Por tratar clarificar este concepto, la IP con la que trabajamos es la respuesta de la consulta DNS y, en definitiva, donde se está tratando de conectar el host mediante la petición.

En último lugar, el elemento final es el controlador. El controlador agrupa varias funciones dentro del escenario. La principal es proveer a la red de inteligencia, toma de decisiones e instalación de las reglas de monitorización. Además, es en este punto donde se indica al *switch* que debe replicar todo el tráfico por el puerto de SPAN. A su vez, en el controlador se ejecuta una segunda instancia, que actúa como colector de la información extraída por el *sniffer*. Esta segunda instancia es la que nutrirá el dashboard para la representación de los resultados y al controlador de las reglas necesarias que se han aprendido mediante la captura y catalogación de los paquetes por host. Para realizar la instalación se utiliza la herramienta CURL. En el anexo B se muestran algunos ejemplos de cómo se instalan las reglas según el matcheo de los paquetes.

Para la representación de los resultados se han examinado algunas bases de datos. La que más se adecúa las necesidades del trabajo es influxDB. Es una base datos basado en series temporales y está perfectamente integrada y documentada con Grafana que se ha seleccionado para la representación de los datos.

A continuación, se muestra esquematizado el controlador con las dos instancias anteriormente mencionadas:

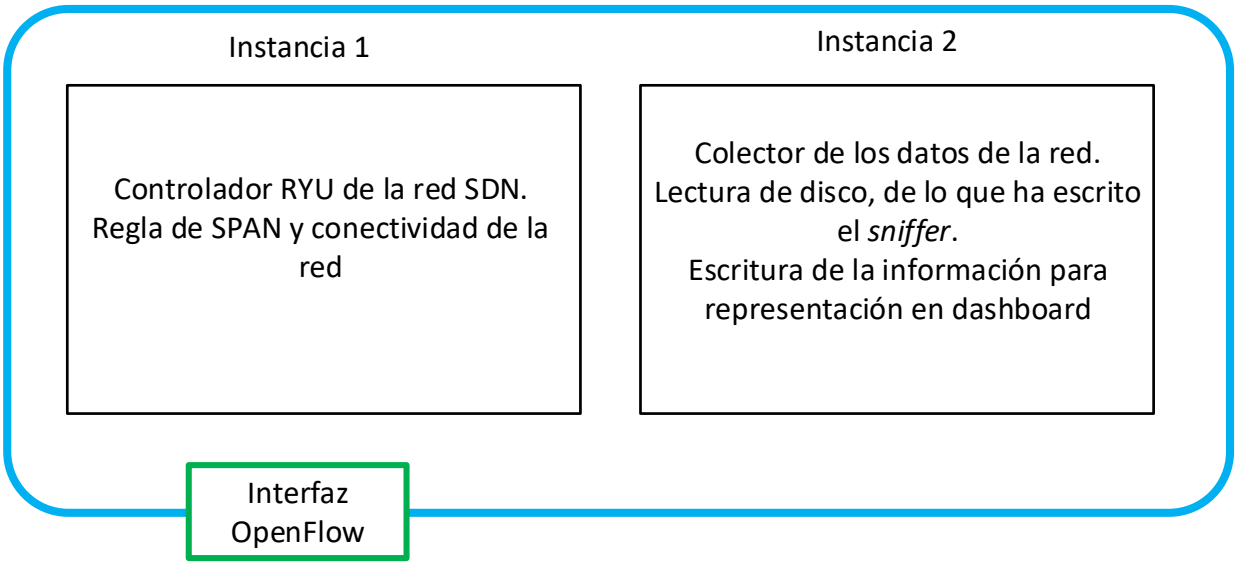


Figura 3-5: Diseño controlador y colector del escenario a alto nivel

Los datos que se reciben en el controlador y que se procesan son, por cada paquete, IP origen y destino, puerto origen y destino, el nombre del host al que se conecta y el tamaño en bytes de cada paquete. El colector lo que realiza es una agrupación de paquetes por flujo y va incrementando el número de paquetes por flujo según se van categorizando y se van sumando el número de bytes globales dentro de cada flujo. Al final lo que se pretende representar es, por cada flujo identificado, cuantos paquetes y bytes se han capturado y se corresponde bajo la etiqueta del host.

Para finalizar se muestra el flujo completo de intercambio de información durante una ejecución del escenario. A grandes rasgos el flujo seguido es, en primer lugar, se inyecta el tráfico en la red desde el host de la izquierda de la figura inferior.

El tráfico llega al *switch* al que está directamente el host que inyecta el tráfico y el resto de componentes de la red. Estos componentes son, el *sniffer* en la parte inferior del escenario y el controlador en la parte superior del mismo. La clave del escenario reside en la regla definida en el controlador mediante todo el tráfico que entra se replica por el puerto donde está el *sniffer* escuchando y listo para procesar el tráfico.

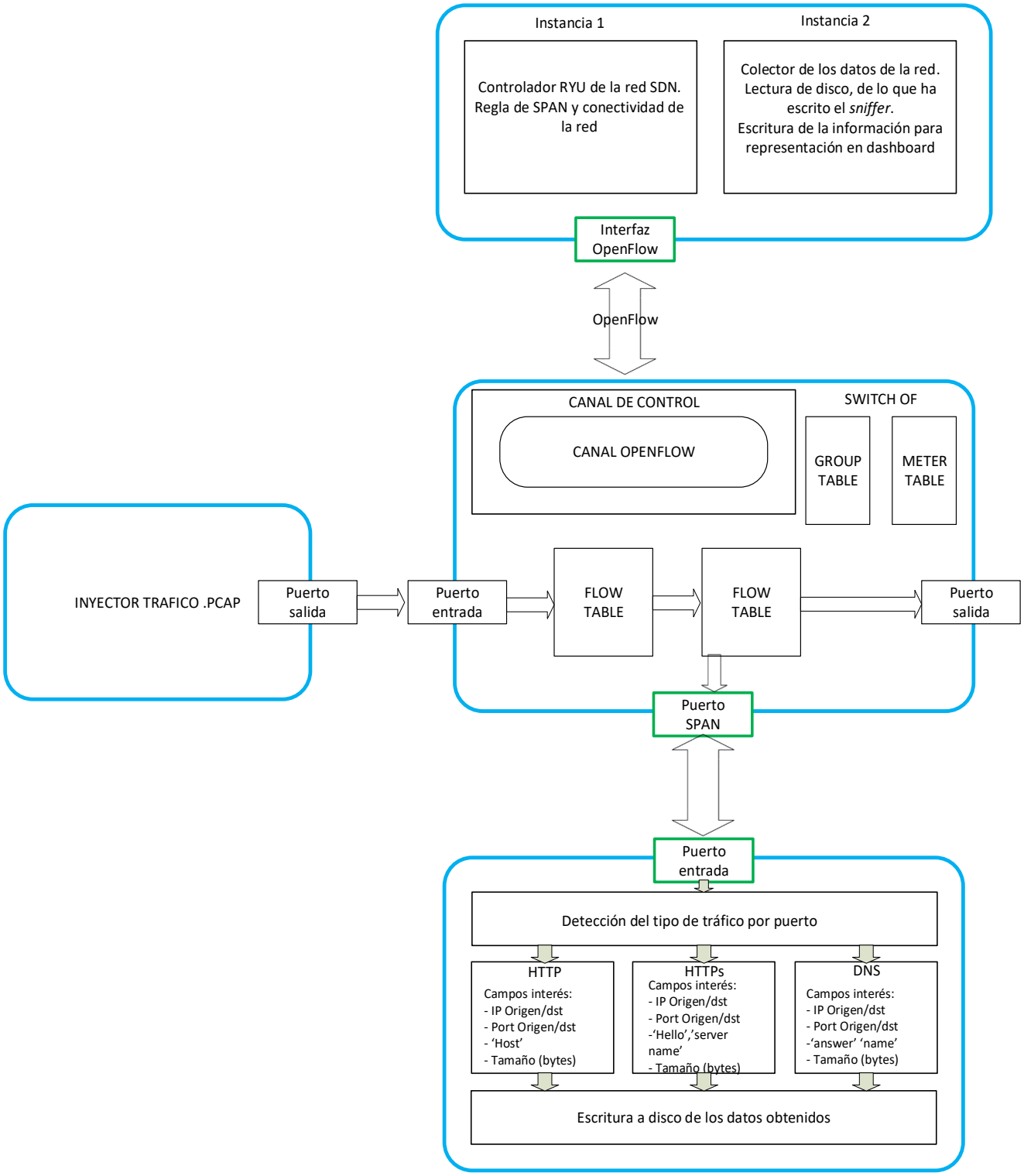


Figura 3-6: Diseño global del escenario a alto nivel

Una vez que se ha procesado y extraído los campos de interés, se escriben en el disco para que la segunda instancia del controlador pueda trabajar con ello y representarlo para apreciar que se está comportando la red. En la siguiente figura se muestra la dependencia de cada uno de los bloques descritos anteriormente en más detalle:

3.3 ***Conclusiones***

En el apartado de diseño se han expuesto las decisiones a la hora de implementar el escenario que vamos a utilizar para poder desplegar el *framework* de monitorización. Así como también se ha descrito el diseño de los distintos componentes que forman el escenario con sus funcionalidades e intercambio de mensajes.

La importancia del controlador es vital, ya que es el que más se ajusta y facilita las tareas de implementación y monitorización del escenario. También que este implementado en Python ha propiciado su elección.

En cuanto a la decisión de la herramienta que soporta el escenario de los tres que se barajaban se optó por Mininet por la experiencia que ya se tiene sobre él de trabajos pasados y porque para el estudio propuesto cumple perfectamente con los requerimientos impuestos.

InfluxDB es claro candidato como base de datos debido a la elección del software de representación y a su buena integración con el mismo.

4 Desarrollo

4.1 *Introducción*

En la parte de desarrollo, se va a explicar el proceso seguido para desarrollar el monitorizador, modificar el controlador para programar que el escenario funcione como deseamos, la implementación de un colector de datos, que los recibe desde el *sniffer*, para agruparlos y darles el formato adecuado para luego escribirlos en una base de datos y finalmente representarlos. Todo el código implementado y referente a este trabajo de fin de Máster esta subido en un repositorio de GitHub [41].

4.2 *Sniffer*

Para el desarrollo del *sniffer* de paquetes que atraviesan la red, se ha utilizado el lenguaje de programación C. El *sniffer* está basado en la librería de C *pcap*.

El funcionamiento es el siguiente, el *sniffer* está escuchando por el puerto del *switch* ejecutando *OpenFlow* al que está conectado, y este puerto está configurado como SPAN, es decir, todo el tráfico que atraviesa el *switch* lo replica por dicho puerto, aunque no vaya destinado a él.

Para aportar mayor granularidad al *sniffer* se ha implementado de tal manera que se pueden aislar las diferentes pruebas, es decir, se ha programado un filtro que permite filtrar el tráfico por puerto. Es útil para poder segmentar el tráfico de la red y centrar el estudio a cierto tipo de tráfico. Los tipos de tráfico que vamos a monitorizar principalmente son, HTTP [42], DNS [43] y HTTPs [44], puertos 80, 53 y 443 respectivamente.

Una vez que se conoce como llega el tráfico al *sniffer* y qué tipo de tráfico se va a monitorizar vamos a entrar en detalle de la implementación del *sniffer*. Cabe destacar que para la implementación ha sido necesario estudiar y analizar minuciosamente los protocolos mencionados anteriormente para poder recorrer los paquetes y acceder a los campos de interés.

Para comenzar a implementar el *sniffer* se ha utilizado la función *pcap_open_live* como se ha mencionado al principio. Y se ha seguido los siguientes pasos:

- En primer lugar, es necesario conocer por cual interfaz de HMON se va recibir el tráfico, para ello se ha implementado de tal manera que el programa detecta automáticamente las interfaces disponibles y elige el mismo por cual se va a recibir el tráfico.
- En siguiente lugar aplicamos los filtros por si queremos separar el tráfico o recibir todo y analizarlo. En este punto es donde se especifica el puerto del que se desea capturar tráfico o, en su defecto, este parámetro se puede configurar como “ip” y se reciben todos los paquetes que atraviesan el *switch*.
- Una vez definidos todos los parámetros iniciales se indica en el programa *pcap* que entre en el bucle principal de ejecución. En este estado, cada vez que recibe un paquete de la red ejecuta las funciones que se han definido en función de qué tipo de tráfico sea.
- Cuándo se han capturado todos los paquetes deseados, el cual es un parámetro que se introduce al principio de la ejecución, se detiene la captura.

- Por último, una vez que el paquete ha pasado por el *sniffer*, este se traslada al colector. Al colector se transfieren los paquetes procesados, se escribe en disco la quíntupla, IP origen, IP destino, puerto origen, puerto destino y protocolo añadiendo la etiqueta de la página visitada.

Para poder determinar el campo de la etiqueta se han desarrollado unos disectores diferenciados por cada tipo de tráfico (HTTP, DNS y HTTPS), en los cuales se recorre el paquete de diferente manera hasta alcanzar extraer los campos de interés.

De manera muy breve y a alto nivel se va a explicar en cada función como se extrae y cuál es el campo o campos relevantes en los que nos centramos. En primer lugar, para HTTP, puesto que se manda toda la información en texto plano y sin cifrar es sencillo, tenemos que recorrer el paquete de red hasta alcanzar el campo host y almacenarlo en una variable que posteriormente se agregara a la quíntupla para enviarla al colector.

Para el segundo protocolo del desarrollo, DNS, se ha requerido de dos funciones principalmente. En este caso tenemos dos campos de interés, el primero de ellos es el host al que se solicita al servidor para que lo traduzca a la IP a la que conectarse, y la IP a la que conecta una vez resuelto el nombre. Para la realización, tras el estudio de los paquetes y el protocolo DNS, diferenciamos entre si es una petición o una respuesta de DNS, son de interés las respuestas DNS ya que ahí se encuentran todos los parámetros necesarios. Una vez que identificamos una respuesta DNS procedemos a inspeccionar el paquete para almacenar los campos de “name” y “address”. Una vez adquiridos todos los datos conformamos la quíntupla más la etiqueta para enviarla al colector para que lo agrupe por flujos y así poder representar los datos.

En último lugar queda por analizar el proceso de inspección del paquete HTTPS. Este tipo de paquetes son los más peculiares de los tres que hemos analizado ya que es necesario tener varios factores a la hora de tratar con ellos. Como breve introducción se sabe que el tráfico HTTPS usa SSL (Secure Sockets Layer) y está encriptado para aportar privacidad y seguridad a las conexiones. Para poder identificar el host al que los usuarios se están conectando con el fin de conocer el tipo de tráfico de la red y tomar en decisiones de ellas, es necesario capturar los paquetes de “Client Hello”. Una vez capturados estos paquetes, podemos acceder al campo de “server name” mediante inspección del paquete y obtener la página o servicio al que se está accediendo.

Tras haber recopilado las distintas quíntuplas más el sexto campo de la etiqueta, es necesario transmitir la información al colector. Para ello, se ha realizado mediante escritura en disco por parte del *sniffer* en C y el colector puede acceder a él cuando sea necesario.

Otra manera de poder realizar el intercambio de información puede ser mediante el diseño cliente-servidor, donde el colector actúa como servidor y el *sniffer* como cliente. Los datos capturados en el *sniffer* se envían en tiempo real hacia el colector, y se puede imponer una premisa de *timeout* en el colector que si pasan 7 segundos sin recibir tráfico cierre la conexión.

4.3 Colector

El denominado colector está desarrollado íntegramente en Python. La función principal que tiene asignada es recibir los datos extraídos por el *sniffer* y agruparlos para posteriormente representarlos y poder obtener conclusiones.

En la implementación, en primer lugar, se indica el nombre del archivo del que debe leer los datos y los posibles parámetros necesarios, como pueden ser delimitadores… Una vez que se ha definido el archivo de lectura, el siguiente paso es crear la base de datos donde vamos a guardar los datos para posteriormente, representarlos usando Grafana. La base de datos utilizada es InfluxDB debido a la fácil integración que tiene con la herramienta de representación de los resultados utilizada y también porque guarda todos los datos como series temporales, un factor importante a tener en cuenta.

Cada vez que se recibe una quíntupla con la etiqueta se va agrupando por flujos, en el caso de que existan flujos sin etiqueta, que puede suceder, permanecen a la espera hasta que llegue una quíntupla con etiqueta y se actualiza el nombre del host automáticamente. Durante la agrupación en flujos se comprueba, la IP origen, IP destino, puerto origen y puerto destino. El colector nutre al controlador con la información necesaria para poder establecer políticas de switching en base al tráfico que se ha detectado en la red, dicha información es sobre la información de flujos que si aporta *OpenFlow* en su implementación. Los datos que nos aporta *OpenFlow* se aprovechan para enriquecer constantemente la información obtenida. En concreto esta información es la IP origen y destino, el puerto origen y destino, el número de paquetes por flujo y el número de *Bytes*. A continuación, se muestra una figura que ilustra la información que se puede solicitar a al switch *OpenFlow*:

Figura 4-1: Ejemplo consulta de las *flow stats* mediante CURL formateadas en JSON

Para los distintos flujos de tráfico, DNS, HTTP, y HTTPs tienen algunos campos que comparten, como pueden ser los campos de IP origen, destino o puerto origen y destino, también el campo de host, aunque se obtiene de manera en cada caso. Existen otros campos que son específicos de cada protocolo. Estos campos diferenciadores se utilizan para extraer la información sobre el host y filtrar mensajes que no tienen información relevante. Por ejemplo, en el caso de DNS, se filtra para inspeccionar aquellos paquetes que sean *answer* ya que necesitamos extraer el host y la IP que conecta. En el caso de las *queries* aparece el host al que nos queremos conectar, pero no la IP que resuelve, porque precisamente es lo que se está consultando. En el caso de HTTPs, hay que capturar el paquete de *Hello* para poder capturar el resto de los paquetes del flujo. Del paquete que

contiene el mensaje de *Hello* se puede extraer el host y de esta manera etiquetar el resto de paquetes que compartan IP, puerto de origen y destino. Si se pierda el paquete de clave, por las características de cifrado de HTTPs no se puede catalogar el resto de paquetes.

Cuando todos los parámetros mencionados, coinciden se va incrementando el campo de número de paquetes del flujo, y también se va sumando el número de Bytes. Que es un campo transmitido desde el *sniffer*. A medida que se van generando los flujos, estos se formatean para adecuar los datos a las restricciones que impone la base de datos, en nuestro caso InfluxDB. Se han definido los siguientes campos en la base de datos, IP origen, IP destino, puerto origen, puerto destino, número de Bytes, número de paquetes y host del flujo.

En el caso de que se requiera hacer mediante sockets se puede configurar un *timeout* mediante el cual si el colector no recibe información durante 7 segundos cierra la conexión con el monitorizador. Este tiempo es modificable según las necesidades del experimento bajo estudio. Esta funcionalidad se utilizó en las primeras fases de los experimentos cuando la transmisión de la información se hacía mediante sockets. No se ha eliminado del documento para que conste que se hizo, pero como no era la finalidad del trabajo se optó por escribir a disco los resultados y es por esa vía por donde se ha proseguido.

4.4 ***Inyección tráfico a la red***

Para realizar las pruebas se ha capturado tráfico navegando por páginas de interés para el estudio y filtrando por puerto en Wireshark. Cada captura de Wireshark se ha almacenado en un archivo .pcap, para posteriormente introducirlo en la red mediante el comando *tcpreplay*.

La misión de *tcpreplay*, consiste en transmitir todos los paquetes almacenados en un archivo a la misma velocidad a la que se han ido capturando y almacenando, pero también se puede establecer cómo parámetro una tasa de reenvío distinta a la tasa de captura. Es un parámetro útil en nuestro estudio para observar y probar el rendimiento del *framework* implementado a distintas tasas para comprobar cómo se comporta a tasas altas de transmisión.

4.5 ***SPAN switch***

Uno de los elementos más importantes dentro de nuestro escenario, es el *switch*. El *switch* en SDN, principalmente trabaja en el plano de datos y recibe la inteligencia mediante el controlador de la red al que está conectado.

La importancia del *switch* en nuestro caso concreto es vital, ya que es por donde pasa todo el tráfico que debemos monitorizar, por ello una configuración correcta nos puede facilitar mucho la tarea de capturar el tráfico desde dónde estamos ejecutando el programa *sniffer*.

Para configurar la funcionalidad de SPAN, que el puerto replique todo el tráfico que atraviesa el *switch* por el puerto que queramos, tenemos que acudir al plano de control de la red, es decir, al controlador, que es donde se centraliza la inteligencia de la red. En nuestro caso como hemos comentado anteriormente el controlador es RYU, implementado en Python. Tras realizar un estudio del controlador y las posibilidades que ofrece se ha definido una acción mediante la cual, si el destino no ha sido aprendido en la tabla MAC del *switch* se envía el paquete mediante inundación en la red, pero en el caso que se sepa a donde reenviar el paquete se envíe por el puerto en el que se ha aprendido, pero además se ha añadido que se envíe siempre por el puerto 3 que es donde está conectado y escuchando el *sniffer*.

A continuación, se muestra los tres hosts y el *switch* involucrado en el estudio. Mediante la prueba mostrada en la siguiente figura se puede ver si la acción está configurada en el controlador y cómo afecta al reencaminamiento del tráfico en el *switch*. Las IPs privadas configuradas en los distintos hosts son en H1 10.0.0.1, H2 10.0.0.2 y HMON 10.0.0.3:

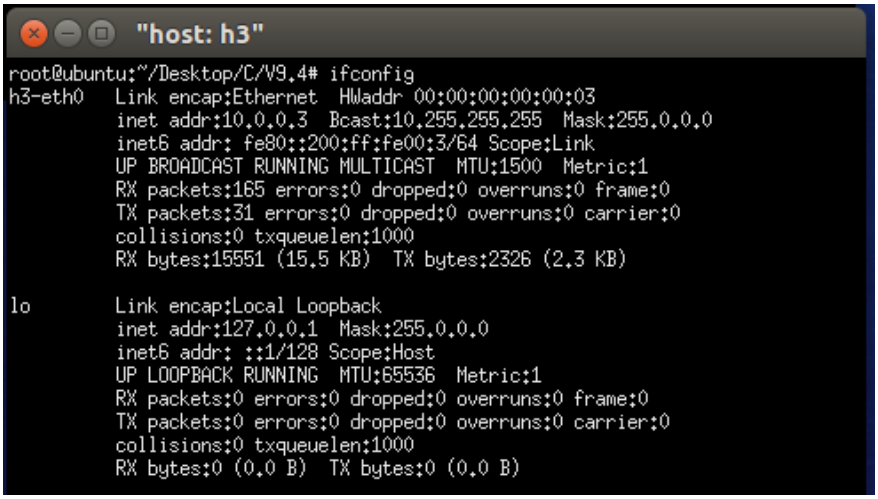


Figura 4-2: Configuración de red del host HMON

Una vez verificada la IP del host HMON, se procede a realizar un ping entre H1 y H2 y, mediante tcpdump, se va a comprobar cómo el host que está conectado el puerto configurado como SPAN está recibiendo el tráfico que realmente está destinado a H2 (IP configurada 10.0.0.2):

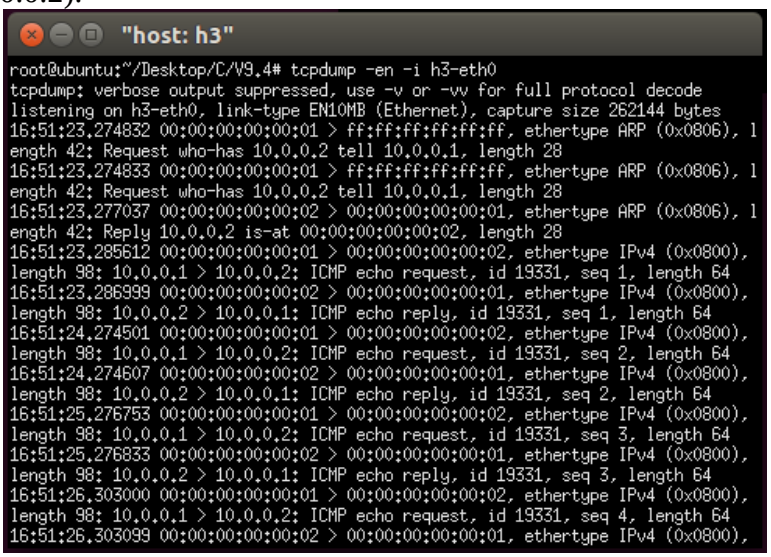


Figura 4-3: Captura de pantalla de tcpdump en HMON

En la figura 4-2 se puede observar como HMON (aparece como h3 en la figura), está recibiendo el tráfico destinado a H2. Este hecho certifica el funcionamiento deseado.

4.6 ***Representación datos***

En cuanto a la herramienta de representación de los datos y de los resultados obtenidos se ha optado por Grafana, que se abastece de la información necesaria mediante la base de datos influxDB que está totalmente integrada con dicha herramienta.

Para la representación de datos se ha optado principalmente por tres tipos de gráficas. Una gráfica tipo tarta con el porcentaje de visita de los distintos hosts de la captura Se adjunta un ejemplo de la gráfica mencionada en Grafana:

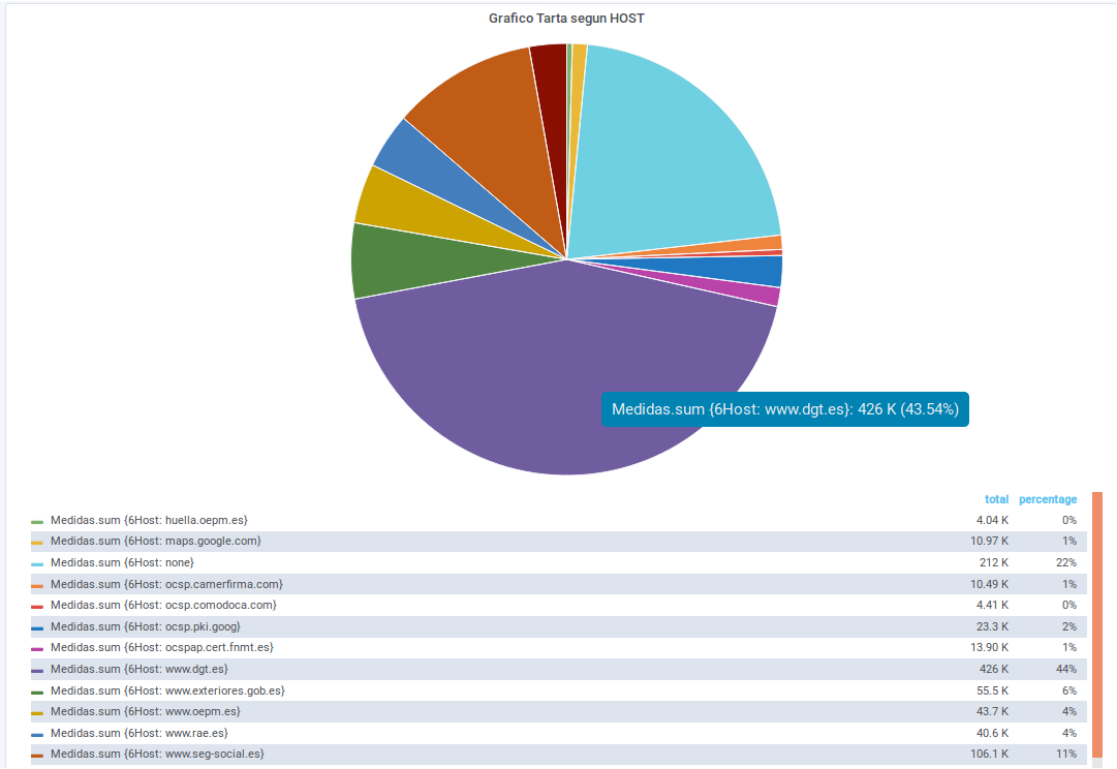


Figura 4-4: Ejemplo grafica de tarta para porcentaje visitas a hosts

Desde Grafana también se representa el número de Bytes y paquetes en los distintos flujos y el total de la captura. A modo de ejemplo, se adjuntan las dos gráficas con el único fin de introducir la herramienta para que posteriormente resulte más familiar al lector.

Figura 4-5: Ejemplo grafica tiempo vs número de paquetes

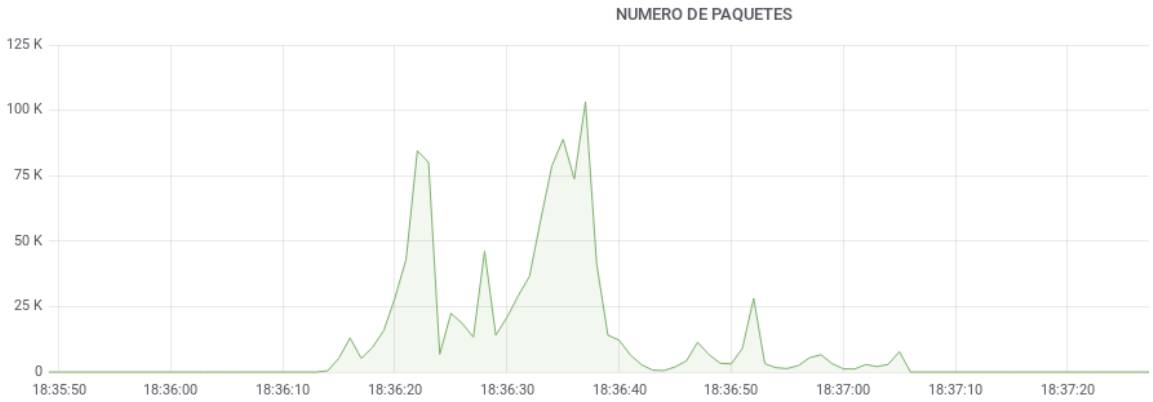


Figura 4-6: Ejemplo grafica tiempo vs número de Bytes

En las dos figuras superiores se puede observar ejemplos de graficas que nos servirán como apoyo para estudiar la eficiencia del desarrollo realizado.

4.7 ***Conclusiones***

En el apartado de desarrollo se han definido y explicado a alto nivel los programas desarrollados, con funcionalidades y puntos clave del diseño. Dichos programas son el monitorizador y el colector. En siguiente lugar se ha detallado la modificación realizada en el controlador para que funcione según las necesidades del estudio. De la misma manera se ha explicado el método mediante el cual se inyecta tráfico a la red para realizar las distintas pruebas y comprobar el correcto funcionamiento de todas las partes del experimento. Y en último lugar se introducido el tipo de gráficas que se van a mostrar y la herramienta sobre la que se representan los resultados con ejemplos.

Todo el código referente a este trabajo de fin de Máster está almacenado en un repositorio de GitHub.

5 Integración, pruebas y resultados

5.1 *Introducción*

Una de las secciones más importantes de este trabajo es el apartado de integración, pruebas y resultados, ya que es en este punto dónde realmente se puede comprobar cómo de bueno y eficiente es el *framework* desarrollado

Para tratar de determinar cómo de eficaz es, se han establecido una serie de pruebas, las cuales, nos servirán como base para poder obtener conclusiones. Para obtener el *ground truth*, que son datos fiables que se han obtenido mediante herramientas del mercado perfectamente testeadas y verificadas, se va a utilizar tshark. En los próximos sub apartados se mostrará cómo se han obtenido dichos valores "objetivos", con los que se comparará y se medirá como de bueno es el *framework* implementado.

Se han definido tres grandes pruebas, las cuales son, una prueba de rendimiento, prueba de error relativo y en último lugar unas comparaciones gráficas. Para tratar de obtener resultados aún más precisos se va a diferenciar en las pruebas por tipo de tráfico para poder dirimir si se puede optimizar más aún la obtención de los parámetros relevantes de la red y ver para que tipo de tráfico funciona mejor el sistema de monitorización.

Para la realización de las pruebas se han obtenido una serie de archivos .pcap mediante la navegación por páginas de interés, los cuales, son inyectados a la red para poder replicar el experimento y apreciar las variaciones.

5.2 *Obtención de ground truth*

El primer paso para evaluar el trabajo realizado es obtener el *ground truth*, para ello utilizamos programa de análisis de tráfico tshark. Tshark es una versión de Wireshark, pero sin GUI. Es muy útil para aquellas situaciones en las que no está disponible o no es necesario un entorno gráfico. Para realizar las capturas se realiza mediante líneas de comandos. El comando en detalle aparece especificado en el anexo B.

A continuación, se indican algunas consideraciones que se han tenido en cuenta para poder capturar el tráfico con Tshark. Para que el comando pueda capturar en la interfaz de red seleccionada es necesario ejecutarlo con permisos de súper usuario. En siguiente lugar con el parámetro –i se indica la interfaz por la que debe escuchar para capturar el tráfico, –T precede a los distintos campos que se quieren extraer de la captura. En este caso, los campos seleccionados precedidos de –e son, el número de trama, la IP origen y destino, el puerto origen y destino, el host al que se está estableciendo la conexión, y el tamaño del paquete TCP. Los siguientes parámetros sirven para poner la cabecera, indicar el separador y otros parámetros para dar formato al archivo exportado que se obtiene como resultado. Destacar que para cada tipo de tráfico se necesitan unos campos u otros, aquí se han descrito los comunes a los otros. En el anexo B están detallados los comandos concretos para cada uno de los protocolos estudiados.

El archivo resultante es un CSV, el cual nos servirá como base para realizar comparaciones con los resultados obtenidos a través de nuestro *sniffer*, de ahí la gran importancia del mismo.

5.3 **Prueba de rendimiento**

Para la prueba de rendimiento se va a utilizar la herramienta disponible en Linux llamada *tcpreplay*, dicha herramienta permite transmitir tráfico de red que ha sido capturado previamente, en nuestro caso, con Wireshark.

Se va a hacer uso de las diferentes funcionalidades que ofrece *tcpreplay*, dichas funcionalidades son, entre otras, variar la velocidad de transmisión de los paquetes, así como también se puede poner un tiempo entre transmisión de paquetes, dependiendo del tipo de prueba que se quiera realizar. En nuestro estudio, como queremos ver el rendimiento del *framework* desarrollado vamos a variar las velocidades a transmisión hasta la máxima tasa posible y de esta manera comprobar hasta que tasa es efectivo y, el *sniffer* implementado, ofrece un comportamiento de detección de tráfico aceptable. Se comparan dichos resultados con los obtenidos con tshark.

Como se ha comentado en la introducción del apartado, se van a realizar las pruebas diferenciando por tipo de tráfico para tratar de aislar si funciona mejor para unos casos u otros. En último lugar se hará una comparativa en conjunto de todo el tráfico. Se tomará como buenos resultados aquellos en los que la tasa de detección de los hosts sea la más similar al *ground truth*.

En primer lugar, se muestra la tasa de acierto, cuando se clasifica e identifica el host correctamente, así como cuando se procesan los paquetes correspondientes de manera correcta. Para realizar esta prueba se ha arrancado el escenario mostrado en la figura 3 -1, en el caso de nuestro *sniffer* el escenario es el que se muestra en la figura, pero en el caso de la comparativa con el *ground truth*, en vez de ejecutar nuestro *sniffer*, se ha ejecutado tshark en HMON. A la hora de transmitir los datos al controlador, para que la prueba sea lo más equitativa posible, se ha escrito la información extraída del *sniffer* en un archivo .csv que posteriormente toma el colector para poder representar los resultados y poder obtener conclusiones. Aunque, también se ha implementado en nuestro *sniffer* que se envíe el tráfico mediante un socket UDP al controlador. La transmisión mediante el socket UDP al final no se ha utilizado puesto que se añadía un punto de transmisión en el que podía haber fallos.

Se ha decidido que ambos tomen la información de un CSV debido a que, tras las pruebas realizadas con el *sniffer* implementado enviando la información mediante el socket UDP, y el *sniffer* de thsark escribiendo en un CSV, se perdían paquetes en la transmisión y entonces la comparación se veía alterada y no era fiel a la realidad. Los resultados mostrados a continuación son tomados en la máxima igualdad de condiciones.

Los pasos a seguir para poder llevar a cabo la primera prueba es necesario seguir los siguientes pasos. Brevemente, puesto que toda la información sobre cómo realizar una ejecución del escenario esta descrita en el anexo C. En primer lugar, se transmite el tráfico a la misma velocidad en la que fue capturada, sin ningún tipo de modificación.

Se indica que se va a transmitir el tráfico por la interfaz eth0 de H1, y el archivo previamente capturado que se va a transmitir. En el lado del *sniffer* implementado se compila el programa implementado y posteriormente se ejecuta.

A la hora de ejecutarlo se declara el número de paquetes que se desean capturar de la red. Una vez adquiridos los datos de la red, es necesario darles sentido y agruparlos según flujos. Esta tarea la realiza el controlador mediante el programa colector implementado en Python.

Tras realizar los pasos anteriores con éxito, se puede visualizar el resultado obtenido en la herramienta Grafana.

Para obtener el *ground truth* para los distintos tipos de tráfico, la parte de *tcpreplay* permanece inmutable, mientras que los comandos introducidos en el *sniffer*, que se ejecuta en el host HMON, se detallan en el anexo B.

A modo de resumen y para no entrar en detalles muy concretos lo que se busca es extraer la información relevante de la captura transmitida y escribirla en un fichero CSV que luego será procesado para agrupar por flujos y representado en Grafana para visualizar las diferencias con el *sniffer* implementado. Dicha información relevante se corresponde en cada caso, DNS, HTTP y HTTPs, con diferentes campos.

Para HTTP lo primero que es necesario hacer es imponer el filtro mediante el cual se realizara una criba del tráfico que no corresponde para este caso. Este filtrado se hace mediante el puerto 80. A continuación se indican los campos que se quieren extraer de la captura y que son almacenado en el archivo CSV, Los campos son, IP origen y destino, puerto origen y destino, el host al que se pretende conectar y el tamaño del paquete TCP. Este formato es el mismo que para el *sniffer* implementado ya que luego se pasan por el mismo colector y deben coincidir los campos. En las tablas de resumen para HTTP los valores que se muestran están en tanto por 1.

En el caso de DNS el primer filtro se realiza mediante el puerto 53. Los campos que se especifican para obtener la información son IP origen y destino, puerto origen y destino, que sea una *answer* de DNS, y el *server name* y la IP que resuelve la consulta y el tamaño del paquete.

En último lugar para HTTPs, se filtra por el puerto 443 en primer lugar. Posteriormente, los campos necesarios de IP, puerto igual que en los dos casos anteriores, también es necesario el tamaño del paquete en bytes para la posterior representación. Los campos específicos de este protocolo son el tipo de *handshake* y el *server name*. En las tablas de resumen para HTTPs los valores que se muestran están en tanto por 1.

Una vez obtenido el tráfico y almacenado en un archivo CSV, se ejecuta el colector que ejecuta las mismas operaciones que el colector anterior, pero varía el nombre de la base de datos en la que almacena los datos puesto que es necesario diferenciar lo capturado con el *sniffer* y con tshark.

5.3.1 **Rendimiento para tráfico HTTP**

Una vez ejecutados ambos colectores ya se puede proceder al análisis de los resultados. A continuación, se muestran las gráficas obtenidas para cada ejecución. En primer lugar, se muestra en la figura 5-1 el resultado de porcentaje de tráfico hacia los distintos Hosts por los que se navegó durante la captura de los paquetes.

5.3.1.1 Tráfico transmitido a velocidad estándar

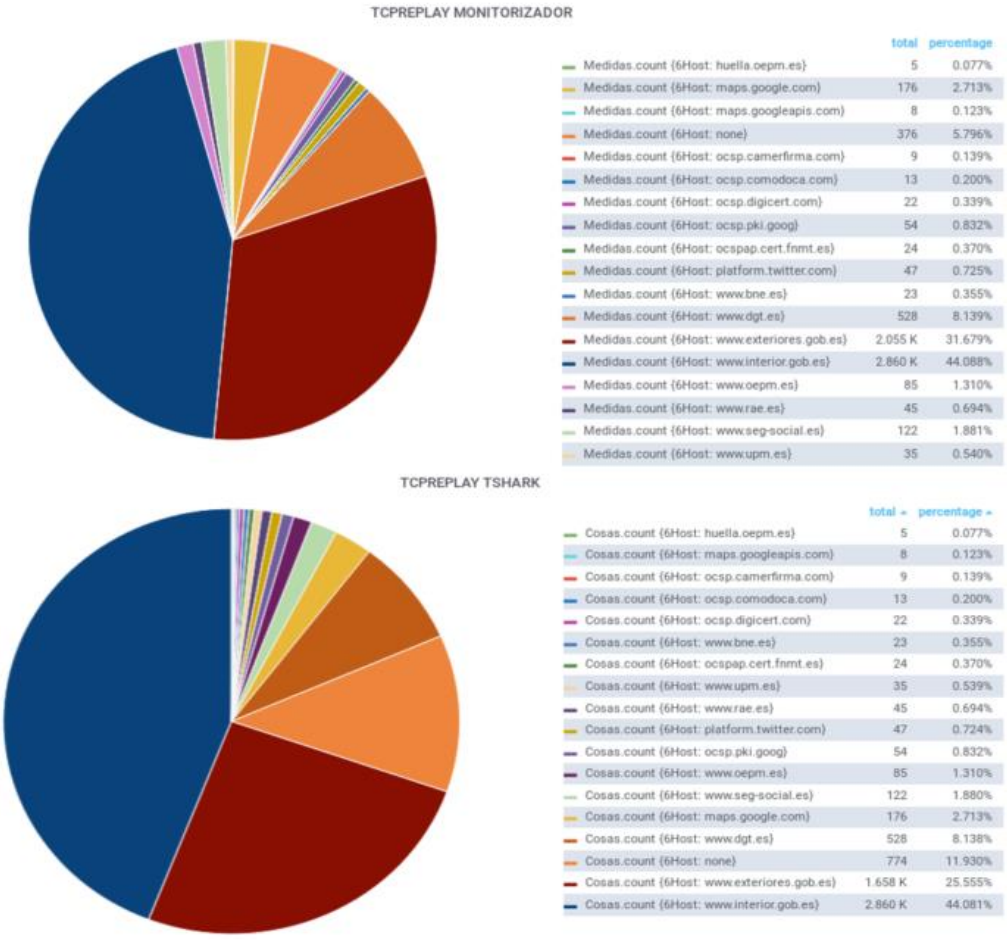


Figura 5-1: Porcentaje visitas HTTP a hosts para velocidad estándar de transmisión

En la primera gráfica de la figura superior se puede apreciar el porcentaje de paquetes que se corresponde con cada host visitado mediante el *sniffer* implementado. Aparece el campo none, el cual se corresponde con aquel tráfico que no se ha podido clasificar, ya que no se corresponde con ningún flujo que se haya etiquetado durante todo el proceso.

En las columnas de la derecha, la primera de ellas muestra el número de paquetes acumulados durante toda la captura agrupado por host, y la segunda representa el porcentaje con respecto al total de paquetes.

En la segunda gráfica de la figura 5-1 se muestra el resultado de la ejecución con tshark.

Tras la primera obtención de los resultados, a la velocidad en la que se produjo la captura se puede observar que el *sniffer*, para tráfico HTTP, funciona bastante bien. Se logran detectar todos los hosts visitados. Se puede afirmar ya que cuando se realizó la captura de

los paquetes se anotaron todos los hosts visitados, y, por otro lado, se confirma con los resultados obtenidos del *ground truth*.
En cuanto al porcentaje de paquetes de cada Host, varía mínimamente y ambos resultados son muy ajustados. Por lo tanto, se puede deducir que el *sniffer*, para tráfico HTTP y para la velocidad estándar de navegación, funciona de manera correcta.

Con el fin de aclarar más los resultados obtenidos anteriormente, se muestra una tabla que recoge, en la primera columna, los distintos hosts detectados durante la transmisión de los paquetes. La segunda y tercera columna se corresponden con el cálculo del error relativo cometido en cuanto número de paquetes y número de Bytes. La fórmula para obtener dichos resultados se detalla en el apartado 5.4:

Host	Error paquetes	Error *Bytes*
huella.oepm	0	0
maps.google.com	0	0
maps.googleapis.com	0	0
ocsp.camerfirma.com	0	0
ocsp.comodoca.com	0	0
ocsp.digicert.com	0	0
ocsp.pki.goog	0	0
ocspap.cert.fnmt.es	0	0
platform.twitter.com	0	0
www.bne.es	0	0
www.dgt.es	0	0,447679325
www.exteriores.gob.es	0,239445115	0,428051002
www.interior.gob.es	0	0
www.oepm.es	0	0
www.rae.es	0	0
www.seg-social.es	0	0
www.upm.es	0	0

Tabla 5-1: Tabla resumen prueba HTTP a 0.33 Mbps

Se puede observar como los errores cometidos se dan en dos flujos. Para el caso del host *www.dgt.es* se han capturado exactamente el mismo número de paquetes, sin embargo, hay diferencia en la cantidad de *Bytes* capturados. Este hecho apunto a un fallo en la detección del *payload* de ese host que ha provocado esa discrepancia. En el caso del segundo host, el error se debe a que paquetes no se han clasificado como deberían lo que deriva en el error cometido tanto en el número de paquetes como en la cantidad de *Bytes.* De manera global, se puede ver que se comete un error considerable únicamente al detectar el número de paquetes del host *www.exteriores.gob.es* pero en el resto de host no. Este hecho puede ser derivado por las características de la propia captura que se está retransmitiendo.

El siguiente paso es forzar el tráfico a velocidades más altas para ver si el *framework* implementado es capaz de obtener resultados favorables y, de esta manera, poder detectar donde se encuentra el límite en cuanto a tasa de paquetes donde sigue dando buenos resultados de clasificación de paquetes y en qué tasa de datos comienza a dejar de ser fiable.

El resumen de la transmisión se de los paquetes mediante *tcpreplay* se encuentra en la siguiente figura:

```
Actual: 6488 packets (4390114 bytes) sent in 102.39 seconds.          Rated: 42876.4 bps, 0.33 Mbps, 63.37 pps
Statistics for network device: h1-eth0
        Attempted packets:         6488
        Successful packets:        6488
        Failed packets:            0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
```

Figura 5-2: Resumen transmisión con *tcpreplay* a velocidad estándar para HTTP

Se han transmitido 6488 paquetes que suman un total de 4390114 Bytes y la duración de la inyección de tráfico es de 102 segundos. La tasa es de 0.33 Mbps y 63.56 paquetes por segundo.

5.3.1.2 Tráfico transmitido a 108.04 Mbps

Para esta prueba se hace uso de una de las opciones de *tcpreplay* en la que se varía la velocidad de la transmisión de los paquetes a la que fueron capturados. En este caso concretamente se va a multiplicar por 1700 la velocidad de transmisión, da como resultado una tasa de 108.04 Mbps. Lo que nos ayudara a ver qué tal se comporta el *framework* a velocidades mayores. Se ha escogido está velocidad de transmisión tras numerosas pruebas ya que, es el umbral en el que el *sniffer* deja de recibir la totalidad de los mensajes transmitidos. De los 6488 paquetes transmitidos se han recibido 6130, es decir, se han recibido el 94,48% de los paquetes.

De la misma manera que en la prueba anterior se comparará con el *ground truth*, que, para este caso, será tomado bajo las mismas condiciones, es decir, trasmitiendo a 1700 veces de la velocidad en el origen. También se comparan los resultados con la transmisión de paquetes a velocidad estándar para ver como evoluciona el aumento de transmisión de paquetes con la detección de paquetes.

Hay que tener en cuenta que, para este tipo de pruebas, tiene efecto el procesador del propio ordenador, así como la carga a la que esté sometido durante la realización de las pruebas. Esta anotación viene por la velocidad de *switching*, que dependiendo a la tasa a la que se reciba tráfico es muy probable que se descarte tráfico. En esta prueba ya se aprecia la perdida de paquetes debido al incremento de la velocidad de transmisión de los paquetes. Los resultados obtenidos se muestran en la figura 5-3.

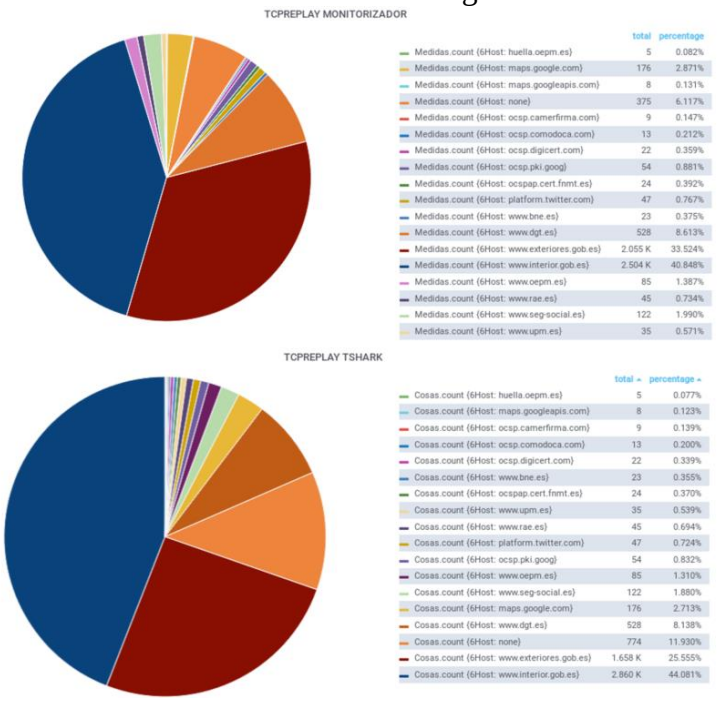


Figura 5-3: Porcentaje visitas HTTP a hosts a 108.04 Mbps

Se puede observar, comparando con la figura 5-2, que la tasa de detección, así como la tasa a la que el *switch* reenvía el tráfico es muy similar. En el caso de la primera gráfica en la figura superior, con el monitorizador implementado, se obtiene una tasa de detección muy similar, lo que deriva que a 1700 veces de la velocidad estándar se comporta de manera satisfactoria. En el caso de tshark se aprecia como hay más paquetes que no se pueden clasificar y han entrado en la parte denominada como "none".

De igual manera que en la prueba anterior y en las próximas pruebas, se va a incluir una tabla resumen de los resultados con la información sobre el error relativo cometido en la captura de los paquetes y el número de *Bytes.* Es una tabla que complementa la información mostrada en la figura 5-2

Host	Error paquetes	Error Bytes
huella.oepm	0	0
maps.google.com	0	0
maps.googleapis.com	0	0
ocsp.camerfirma.com	0	0
ocsp.comodoca.com	0	0
ocsp.digicert.com	0	0
ocsp.pki.goog	0	0
ocspap.cert.fnmt.es	0	0
platform.twitter.com	0	0
www.bne.es	0	0
www.dgt.es	0	0,615384615
www.exteriores.gob.es	0,239445115	0,428051002
www.interior.gob.es	0,124475524	0,165137615
www.oepm.es	0	0
www.rae.es	0	0
www.seg-social.es	0	0
www.upm.es	0	0

Tabla 5-2: Tabla resumen prueba HTTP a 108.04 Mbps

El host *www.dgt.es* sigue la misma tónica que en el caso anterior y se ha sumado un nuevo host en el que se detectan diferencias a la hora de capturar los paquetes y los *Bytes.* El incremento en el error es bajo y centrado únicamente en tres hosts, lo que indica que el rendimiento del framework aún es correcto. En lo respectivo al host *www.exteriores.gob.es* permanece sin cambios con respecto a la prueba anterior.

El resumen de la transmisión de los paquetes mediante *tcpreplay* se encuentra en la siguiente figura:

Figura 5-4: Resumen transmisión con *tcpreplay* a 108.04 Mbps para HTTP

Se han transmitido 6488 paquetes que suman un total de 4390114 Bytes y la duración de la inyección de tráfico es de 0.31 segundos. La tasa de 108.04 Mbps y 20929.03 paquetes por segundo.

5.3.1.3 Tráfico transmitido 159.49 Mbps

En esta prueba se ha dado el salto de velocidad 1700 a velocidad por 2000, lo que hace una tasa de transmisión de 159.49 Mbps. Este salto se debe a que en los pasos intermedios realizados no se detectaba variación notable, los resultados eran similares y no se podían extraer conclusiones de interés. Este paso intermedio se ha realizado para el ver el progreso de la perdida de paquetes hasta llegar al máximo de velocidad de transmisión. Al final del apartado se muestra un gráfico con la progresión de la pérdida de paquetes a medida que aumentaba la velocidad de transmisión.

De la misma manera que en los casos anteriores se comparan los resultados obtenidos con los extraídos bajo las condiciones con el incremento de la tasa de datos y a su vez con los datos obtenidos con tshark.

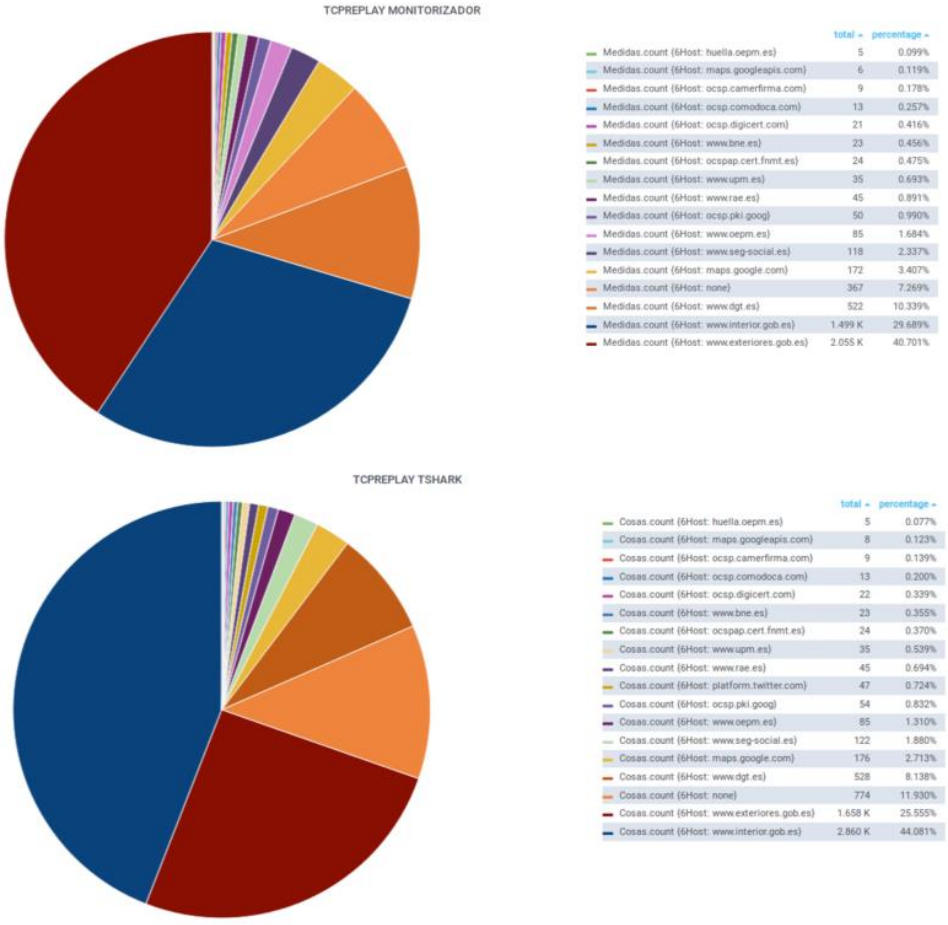


Figura 5-5: Porcentaje visitas HTTP a hosts a 159.49 Mbps

En esta gráfica se ha incrementado el tráfico que se categoriza como *none* ya que no se ha podido procesar e inspeccionar el paquete de manera correcta debido a la elevada carga de datos que se ha sometido al monitorizador. Realmente, el porcentaje de tráfico etiquetado como *none*, se ha incrementado en un 1%, respecto al caso anterior, a 108.04Mbps. Se puede decir que sigue ofreciendo un porcentaje elevado de detección, pese a que no se hayan logrado identificar algunos paquetes.

Si se centra la atención en la tabla resumen que agrupa el error relativo cometido tanto en el número de paquetes como en el número de bytes capturado, se puede apreciar la cantidad de tráfico que no se ha podido catalogar bajo el host correspondiente. A continuación, se muestra la tabla:

Host	Error paquetes	Error Bytes
huella.oepm	0	0
maps.google.com	0,022727273	0,026666667
maps.googleapis.com	0,25	0,251121076
ocsp.camerfirma.com	0	0
ocsp.comodoca.com	0	0
ocsp.digicert.com	0,045454545	0,045454545
ocsp.pki.goog	0,074074074	0,090016367
ocspap.cert.fnmt.es	0	0
platform.twitter.com	X	X
www.bne.es	0	0
www.dgt.es	0,011363636	0,620253165
www.exteriores.gob.es	0,239445115	0,428051002
www.interior.gob.es	0,475874126	0,635321101
www.oepm.es	0	0
www.rae.es	0	0
www.seg-social.es	0,032786885	0,052631579
www.upm.es	0	0

Tabla 5-3: Tabla resumen prueba HTTP a 159.49 Mbps

Se puede apreciar como claramente, para esta velocidad, se empieza a cometer más errores a la hora de catalogar y procesar el tráfico, porque algunos paquetes no llegan a capturarse y se descartan en el monitorizador. Una manera de corregir este efecto sería establecer un búfer para casos de redes de alta transmisión de datos. Es una propuesta que aparece en el apartado de trabajos futuros. Existen dos hosts que tienen una tasa de error elevada, estos hosts son *www.dgt.es* y *www.interior.gob.es*. El error cometido con estos hosts es tan elevado debido al gran volumen de tráfico que corresponde a ellos. Lo que está sucediendo en el *sniffer* es que, durante el procesado de un paquete, se están descartando otros muchos, puesto que al aumentar la tasa los paquetes llegan con mayor frecuencia al *sniffer* y por ejemplo al ser de *dgt.es* llegan uno detrás de otro por lo que se descartan. En el caso del host *platform.twitter.com* aparece marcado con una equis roja porque no se catalogado ningún flujo bajo esa etiqueta. Para ese host se han capturado cero paquetes.

El resumen de la transmisión se muestra a continuación:

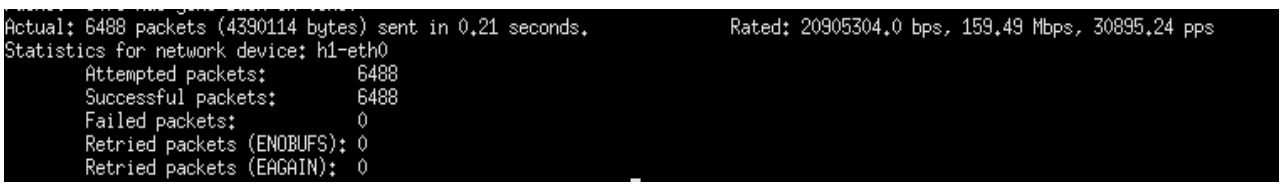


Figura 5-6: Resumen transmisión con *tcpreplay* a 159.49 Mbps para HTTP

Se han transmitido 6488 paquetes que suman un total de 4390114 Bytes y la duración de la inyección de tráfico es de 0.21 segundos. La tasa de 159.49 Mbps y alrededor de 30895 paquetes por segundo.

Teniendo en cuenta estos datos, se realiza una última prueba para ver cómo funciona el monitorizador al máximo de velocidad que permite *tcpreplay* y el hardware en el que se realiza la prueba.

5.3.1.4 Tráfico transmitido al máximo de velocidad

En la última prueba de rendimiento para tráfico HTTP, se retransmite los paquetes capturados con la opción –t de *tcpreplay* la cual permite emitir a la máxima velocidad posible. En este caso la tasa de transmisión obtenida es de 837,35 Mbps.

Esta prueba aportará visión de cómo se comportará el monitorizador implementado para redes de alta velocidad. De la misma manera que en las pruebas anteriores, se compararan los resultados para, al final del capítulo, poder obtener conclusiones fundamentadas.

Bajo estas condiciones, el host emisor transmite 6488 paquetes hacia la red, pero el número de paquetes que el monitorizador es capaz de procesar son 4637. En este escenario el porcentaje de paquetes perdidos es del 28,53%.

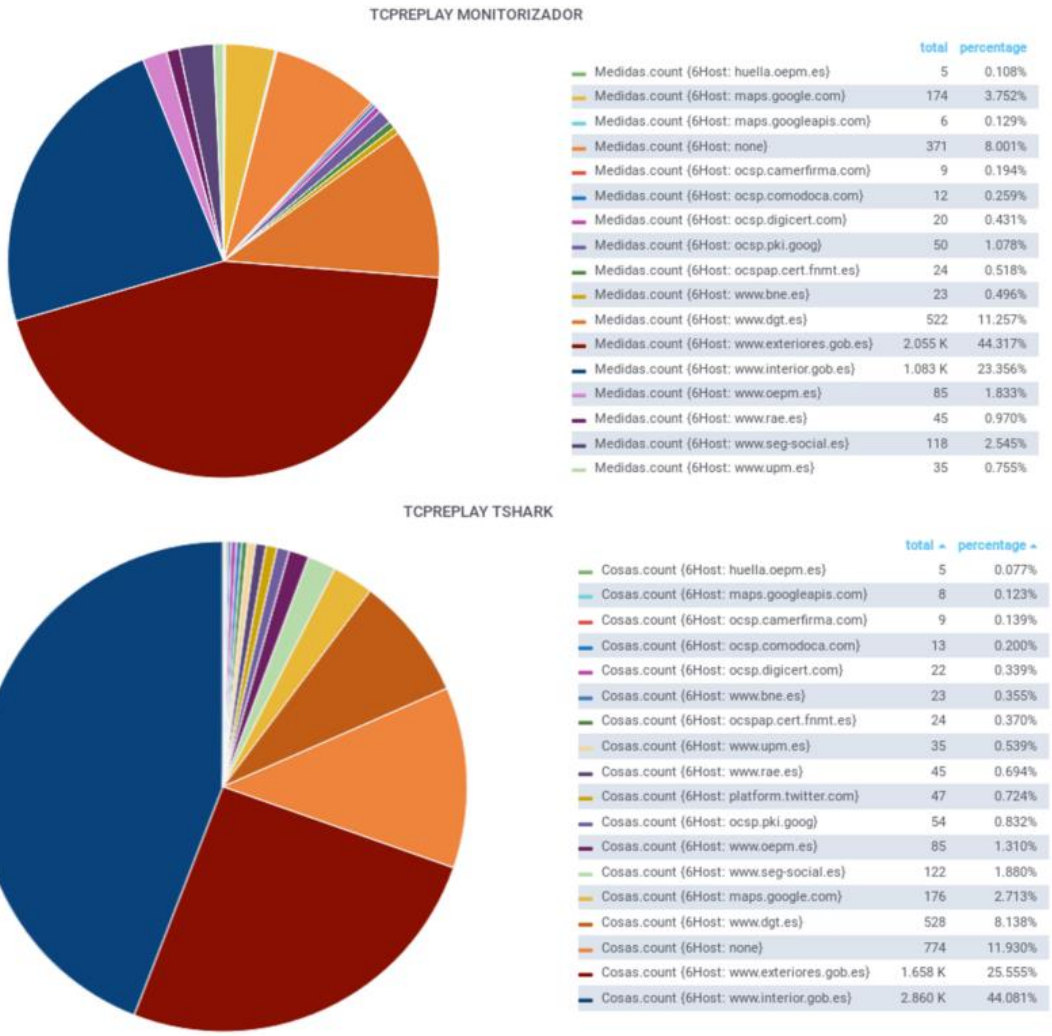


Figura 5-7: Porcentaje visitas HTTP a hosts para 837,35 Mbps

Observando las gráficas de la figura 5-7 es evidente la degradación en el procesamiento de los paquetes. En la gráfica de arriba, con el monitorizador implementado, se aprecia claramente como no se han podido capturar todos los paquetes ya que directamente no aparecen como hosts detectados. Además, dentro de los paquetes que si se logran procesar no se obtienen todos. En cambio, en la gráfica de Tshark sigue mostrando un rendimiento muy bueno con un alto porcentaje de identificación de host, sigue la tónica de las gráficas anteriores.

La tabla resumen del error relativo cometido se muestra a continuación:

Host	Error paquetes	Error Bytes
huella.oepm	0	0
maps.google.com	0,022727273	0,066666667
maps.googleapis.com	0,25	0,251121076
ocsp.camerfirma.com	0	0
ocsp.comodoca.com	0,076923077	0,076923077
ocsp.digicert.com	0,090909091	0,090909091
ocsp.pki.goog	0,074074074	0,090016367
ocspap.cert.fnmt.es	0	0
platform.twitter.com	X	X
www.bne.es	0	0
www.dgt.es	0,011363636	0,620253165
www.exteriores.gob.es	0,239445115	0,428051002
www.interior.gob.es	0,621328671	0,653669725
www.oepm.es	0	0
www.rae.es	0	0
www.seg-social.es	0,032786885	0,052631579
www.upm.es	0	0

Tabla 5-4: Tabla resumen prueba HTTP a 837,35 Mbps

Observando la tabla superior se aprecia como el número de errores se ha visto incrementado en gran cantidad de casos, tanto para el número de bytes como para el número de paquetes. Es un comportamiento fácil de anticipar ya que el monitorizador no puede procesar los paquetes a tasas de transmisión tan altas por lo que se produce descarte de paquetes. De igual manera que en el caso anterior, del host marcado con una equis roja en la tabla no se ha podido identificar ningún paquete durante la captura. La tasa máxima de transmisión ha sido 837,35Mbps.

El sumario de la transmisión a máxima velocidad se muestra a continuación:

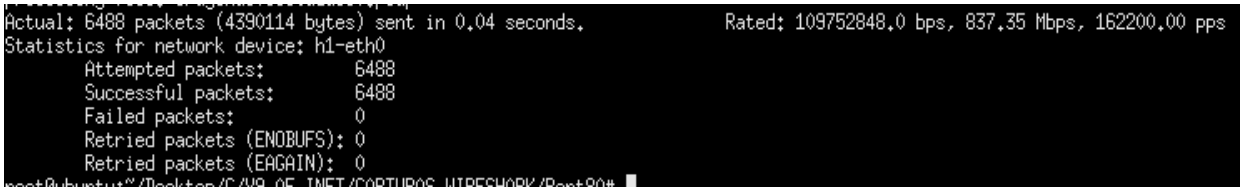


Figura 5-8: Resumen transmisión con *tcpreplay* a máxima velocidad para HTTP

Se han transmitido 6488 paquetes que suman un total de 4390114 Bytes y la duración de la inyección de tráfico es de 0.04 segundos. La tasa de 837,35 Mbps y alrededor de 162200 paquetes por segundo, según se pude observar en la figura 5-8.

5.3.1.5 Resultados globales

En último lugar se van a presentar unas gráficas que pretenden resumir y unificar los resultados extraídos en las pruebas anteriores.

A continuación, se muestra una gráfica que relaciona el aumento de velocidad de transmisión con las pérdidas de paquetes que se producen en el monitorizador porque no es capaz de procesar todos los paquetes que está recibiendo a distintas velocidades.

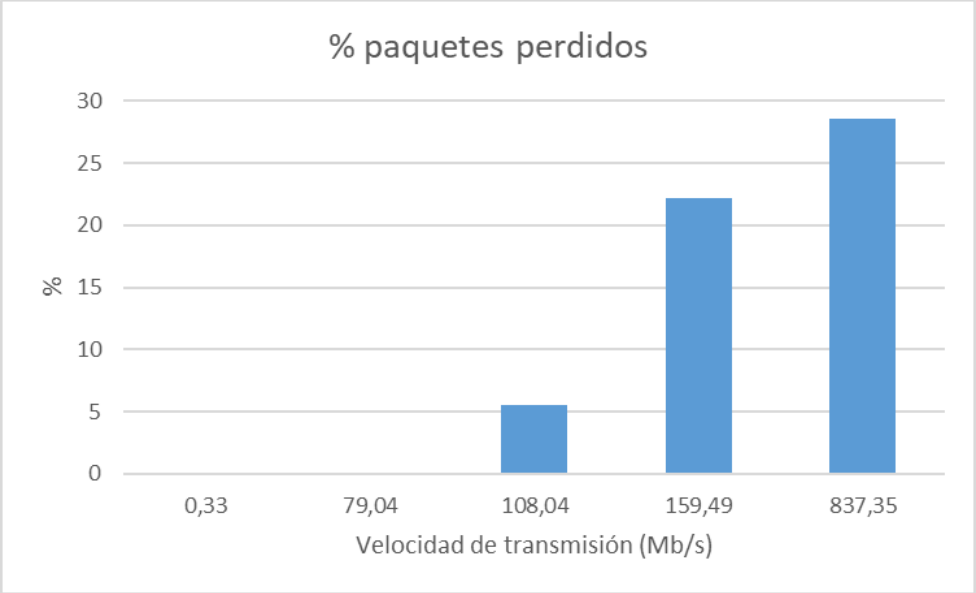


Figura 5-9: Resumen porcentaje paquetes perdidos para HTTP

En la figura 5-9, se puede apreciar, como cabía esperar, que el rendimiento del monitorizador fuera decreciendo a medida que se incrementa la tasa de transmisión de datos por parte del emisor. A máxima velocidad de transmisión el monitorizador presenta una tasa de pérdida de paquetes próximo al 30%. Dicho porcentaje es elevado ya que se ha fijado el umbral en el 15% para estas pruebas. Por lo que se puede decir que a tasas altas no presenta un buen rendimiento.

Para finalizar, se muestra el gráfico que agrupa el porcentaje de bytes perdidos durante la captura a las distintas velocidades indicadas en las pruebas superiores.

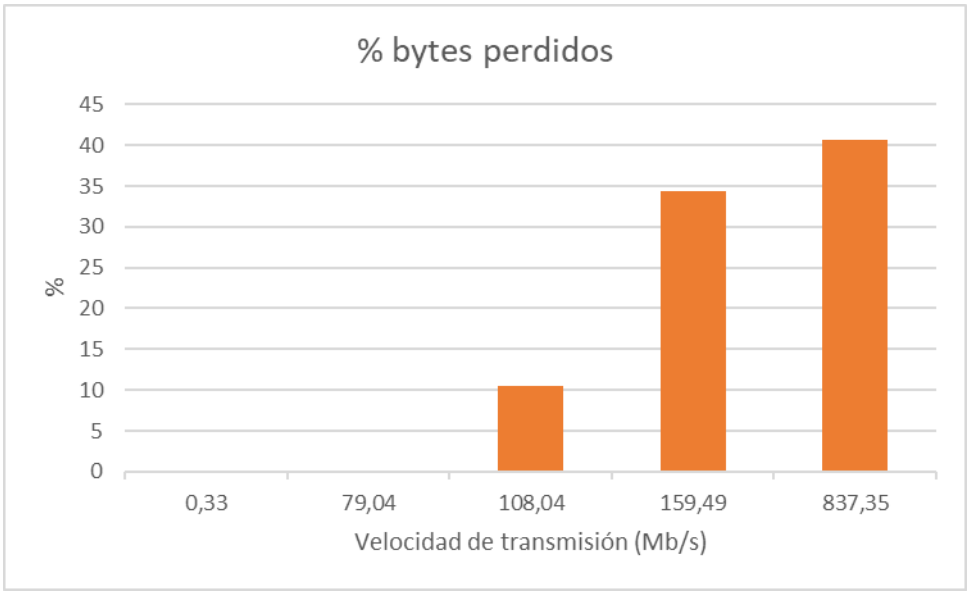


Figura 5-10: Resumen porcentaje bytes perdidos para HTTP

Se puede observar que sigue una tónica similar a la gráfica de paquetes perdidos pero los porcentajes son mayores. Esto muestra que, debido al tamaño variable de los paquetes, se pueden perder paquetes que tengan más tamaño y repercuta en mayor sobre el porcentaje global, o por el contrario que se pierdan muchos paquetes de pequeño tamaño debido a los tiempos de procesamiento de los paquetes sea elevado y no se libere el proceso para seguir capturando los paquetes. Hay que tener en cuenta que el volumen de bytes es mucho mayor que el volumen de paquetes de la captura. Lo que esta gráfica representa que ante el porcentaje de paquetes perdidos implica que la cantidad de bytes que se pierde es considerable. Esto es debido al *sniffer* y no es capaz de procesar todos los paquetes a velocidades elevadas y se producen descartes.

5.3.2 **Rendimiento para tráfico DNS**

Una vez repasado los resultados obtenidos con el monitorizador filtrando únicamente tráfico HTTP, se va a continuar con el rendimiento para tráfico DNS. Para ello, durante la captura se configurado un filtro que descarta todo el tráfico excepto el que tiene como origen o destino el puerto 53.

Para el caso de DNS, esta prueba no se puede realizar de la misma manera que la anterior, puesto que, en esta ocasión, estamos analizando tráfico de DNS y lo que se desea obtener es el host al que conecta y la IP que resuelve la consulta. Es por ello, que la manera de representar los resultados difiere con respecto al apartado anterior. En este caso se va a valorar la capacidad del monitorizador implementado para extraer la IP y el host que resuelve la consulta DNS. Siempre comparando con la información de *ground truth*.

Para esta prueba se va variar la velocidad de transmisión de las capturas y se va a capturar el tráfico con el monitorizador implementado y se comparara con el resultado obtenido con tshark para poder determinar la tasa de detección satisfactoria y las limitaciones del monitorizador implementado.

5.3.2.1 Tráfico transmitido a velocidad estándar

Como se ha indicado en la introducción de la prueba de DNS, en esta ocasión no consiste en cuantificar la cantidad de paquetes identificados correctamente para cada host. En este caso, debido a la implementación realizada, solo se examinan los paquetes que tienen el campo *answer* con valor 1. En los paquetes que cumplen esa condición se inspecciona el host al que se conecta y la IP que resuelve la consulta. A continuación, se muestra la gráfica de porcentaje de detección, contrastándolo con el *ground truth.* El *ground truth* se ha extraído en las mismas condiciones sólo que utilizando tshark, en vez del monitorizador implementado.

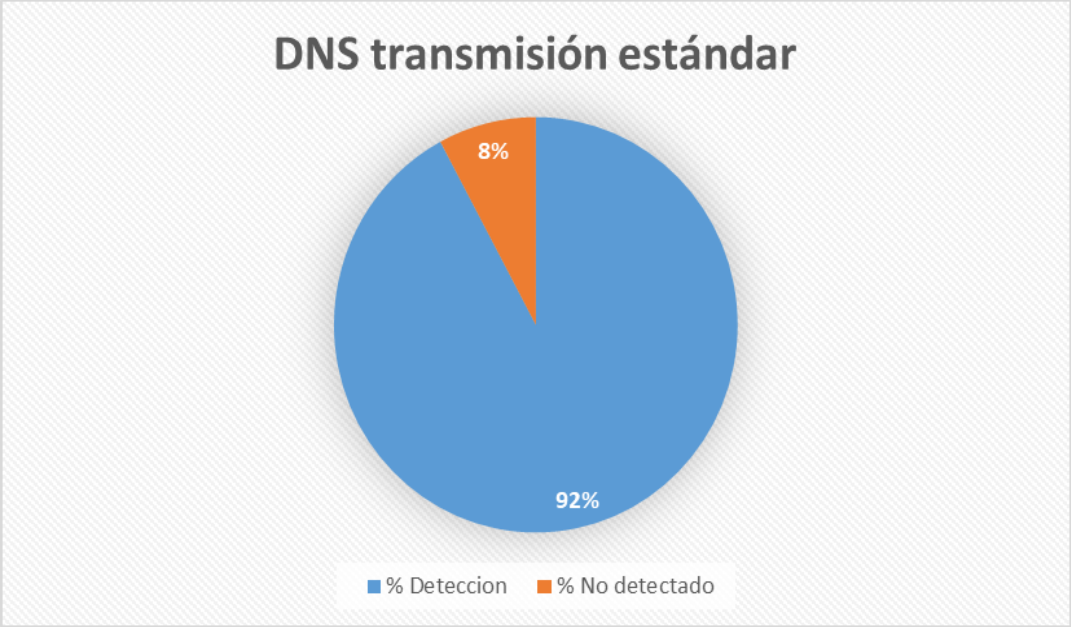


Figura 5-11: Resultados detección DNS. Transmisión a velocidad estándar

Para obtener la gráfica superior se ha ejecutado respectivamente el monitorizador y tshark, se han trasladado los resultados a Excel y se han comparado los resultados. En la captura se han transmitido un total de 1080 paquetes, de los cuales, en el monitorizador

implementado se han detectado como *answer* 482 paquetes y se ha identificado host e IP que resuelve la petición DNS. En el caso de tshark, se han detectado como *answer* 523 paquetes con su respectivo host e IP. En vista de estos resultados, el monitorizador, ofrece un porcentaje de detección de tráfico del 92%. La diferencia que se aprecia entre los resultados obtenidos mediante el monitorizador y tshark es porque, para el caso del monitorizador se ha impuesto una condición de que es necesario detectar los campos de IP y de host para escribirlo en disco. Por ello, si se detecta host pero no la IP, ya no se escribe en el disco para procesarlo después. Tras analizar los resultados de tshark se ha observado que hay un número de host de los que no se escribe la IP que resuelve la consulta DNS en el .csv, únicamente el host, pero realmente sí que resuelve la IP puesto que en la línea superior aparece el host y la IP. Por eso se produce esa diferencia en los resultados. Es un porcentaje elevado de detección, por lo que se puede determinar que funciona de manera correcta a velocidades de transmisión estándar.

El resumen de la transmisión de los paquetes mediante *tcpreplay* se muestra a continuación:

Figura 5-12: Resumen transmisión con *tcpreplay* a velocidad estándar para DNS

Se han transmitido 1080 paquetes que suman un total de 139345 Bytes y la duración de la inyección de tráfico es de 135,73 segundos. La tasa de 0.01 Mbps y alrededor de 8 paquetes por segundo.

5.3.2.2 Tráfico transmitido al máximo de velocidad

En esta prueba se han realizado variaciones intermedias en la inyección de tráfico, pero no se van a mostrar ya que no aportan información relevante. Este caso de estudio, con las capturas de los paquetes iniciales y, para el caso de máxima velocidad de transmisión, tampoco se aprecian cambios en la capacidad de detección del monitorizar, comparándolo con el caso de velocidad estándar. Se omiten los resultados obtenidos porque son idénticos que el caso evaluado anteriormente. Se pueden ver en la figura 5-11.

A continuación, se expone las estadísticas de la transmisión:

Figura 5-13: Resumen transmisión con *tcpreplay* al máximo de velocidad para DNS

Se han transmitido 1080 paquetes que suman un total de 139345 Bytes y la duración de la inyección de tráfico es de 0.001 segundos. La tasa de infinito Mbps y de infinitos paquetes por segundo. Estos resultados los aproxima debido a que la captura no es de gran tamaño y se transmite instantáneamente.

Las conclusiones que se pueden obtener es que en este caso el *sniffer* funciona de manera correcta en el entorno de pruebas realizado, a todo tipo de velocidades. La tasa de no detección es baja de manera global.

5.3.3 **Rendimiento para tráfico HTTPs**

Una vez analizado el rendimiento para tráfico DNS se procede al análisis de los resultados obtenidos para HTTPs. A continuación, se muestran las gráficas obtenidas para cada ejecución variando la velocidad de transmisión. Al final del aparatado se muestra una gráfica que aporta, de manera gráfica, como se ha ido degradando el porcentaje de identificación y captura de paquetes a medida que la velocidad de la transmisión se va incrementando.

5.3.3.1 Tráfico transmitido a velocidad estándar

En la primera ejecución se transmite el tráfico mediante *tcpreplay* a la misma velocidad a la que fue capturado. Dicha velocidad estándar son 0.59Mbps. Los resultados de esta ejecución son los siguientes:

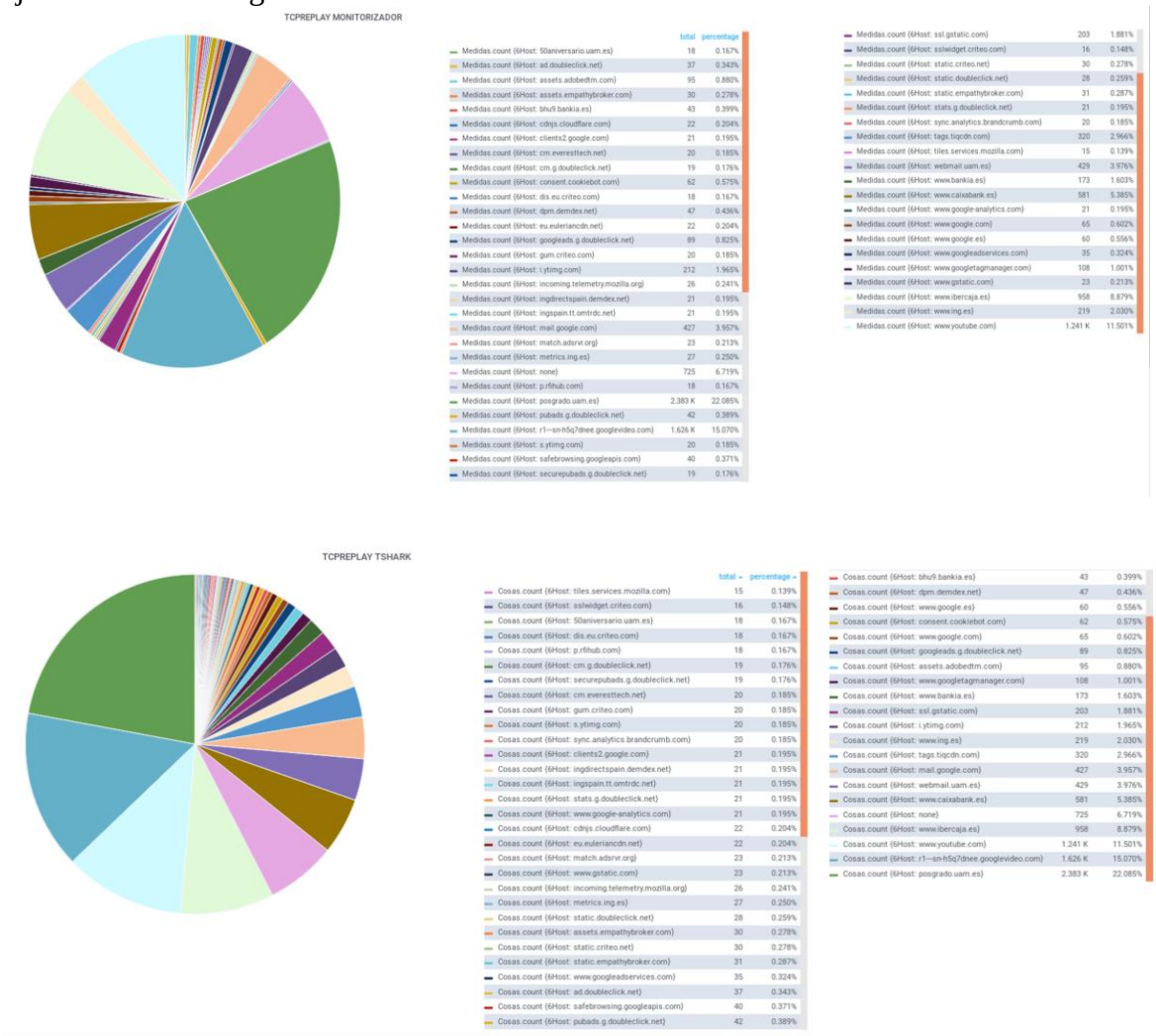


Figura 5-14: Porcentaje visitas a hosts para HTTPs velocidad estándar, 0.59Mbps

En la primera gráfica de la figura 5-14 se puede apreciar el porcentaje de paquetes que se ha identificado con cada host visitado mediante el monitorizador implementado. Aparece el campo none, el cual se corresponde con aquel tráfico que no se ha podido clasificar, ya que no se corresponde con ningún flujo que se haya etiquetado durante todo el proceso.

En las columnas de la derecha, la primera de ellas muestra el número de paquetes acumulados durante toda la captura agrupado por host, y la segunda representa el porcentaje con respecto al total de paquetes.

En la segunda gráfica de la figura 5-14 se muestra el resultado de la ejecución con tshark.

Tras la primera obtención de los resultados, a la velocidad en la que se produjo la captura se puede observar que el monitorizador, para tráfico HTTPs, ofrece una buena tasa de detección. Se logran detectar todos los hosts visitados. Se puede afirmar puesto que cuando se realizó la captura de los paquetes se anotaron todos los hosts visitados. Por otro lado, se confirma que los resultados son buenos comparándolo con el *ground truth* ya que coinciden en su totalidad.

En cuanto al porcentaje de paquetes de cada host, no varía en la captura obtenida mediante el monitorizador y el *ground truth*. Ambos resultados son idénticos. Por lo tanto, se puede deducir que el monitorizador, para tráfico HTTPs y para la velocidad estándar de navegación, funciona de manera correcta y aporta una tasa de identificación buena.

Para arrojar más luz y agrupar los resultados en una sola tabla, se ha realizado la siguiente tabla donde se muestra el error relativo cometido en la captura de los paquetes y de los bytes. La fórmula para obtener dichos resultados se encuentra en el apartado 5.4.

Host	Error paquetes	Error Bytes
50aniversario.uam.es	0	0
ad.doubleclick.net	0	0
assets.adobedtm.com	0	0
assets.empathybroker.com	0	0
bhu9.bankia.es	0	0
cdnjs.cloudflare.com	0	0
clients2.google.com	0	0
cm.everesttech.net	0	0
cm.g.doubleclick.net	0	0
consent.cookiebot.com	0	0
dis.eu.criteo.com	0	0
dpm.demdex.net	0	0
eu.euleriancdn.net	0	0
googleads.g.doubleclick.net	0	0,42
gum.criteo.com	0	0
i.ytimg.com	0	0
incoming.telemetry.mozilla.org	0	0
ingdirectspain.demdex.net	0	0
ingspain.tt.omtrdc.net	0	0
mail.google.com	0	0
match.adsrvr.org	0	0
metrics.ing.es	0	0
p.rfihub.com	0	0
posgrado.uam.es	0	0
pubads.g.doubleclick.net	0	0
r1---sn-h5q7dnee.googlevideo.com	0	0
s.ytimg.com	0	0
safebrowsing.googleapis.com	0	0
securepubads.g.doubleclick.net	0	0
ssl.gstatic.com	0	0
sslwidget.criteo.com	0	0
static.criteo.net	0	0
static.doubleclick.net	0	0
static.empathybroker.com	0	0
stats.g.doubleclick.net	0	0
sync.analytics.brandcrumb.com	0	0
tags.tiqcdn.com	0	0
tiles.services.mozilla.com	0	0,37
webmail.uam.es	0	0
www.bankia.es	0	0
www.caixabank.es	0	0
www.google-analytics.com	0	0
www.google.com	0	0
www.google.es	0	0
www.googleadservices.com	0	0
www.googletagmanager.com	0	0
www.gstatic.com	0	0
www.ibercaja.es	0	0
www.ing.es	0	0
www.youtube.com	0	0

Tabla 5-5: Tabla resumen prueba HTTPs a 0,59 Mbps

En vista de los resultados de la tabla superior se puede apreciar como el sistema del monitorizador implementado para el tráfico HTTPs a la misma velocidad a la que se realizó la captura es muy buena. No se aprecia apenas desviación en el número de bytes ni en el número de paquetes clasificados.

El siguiente paso es forzar el tráfico a velocidades más altas para ver si el *framework* implementado es capaz de obtener resultados favorables sin degradar mucho la detección y, de esta manera, poder detectar donde se encuentra el límite tasa de paquetes donde sigue dando buenos resultados de clasificación de paquetes y en qué tasa de dato comienza a dejar de ser fiable.

El resumen de la transmisión se de los paquetes mediante *tcpreplay* se encuentra en la siguiente figura:

Figura 5-15: Resumen transmisión con *tcpreplay* a velocidad estándar para HTTPs

Se han transmitido 10790 paquetes que suman un total de 6557495 Bytes y la duración de la inyección de tráfico es de 85.30 segundos. La tasa es de 0.59 Mbps y de 126.49 paquetes por segundo.

5.3.3.2 Tráfico transmitido a 161,39 Mbps

Se ha ido incrementando progresivamente la velocidad de transmisión bajo el escenario de estudio hasta llegar a una velocidad 1500 mayor a la transmisión, lo que se traduce en una tasa de 161,39 Mbps. Es en este punto donde se ha detectado una ligera degradación en el proceso de los paquetes debido a la alta tasa en la que estaban siendo recibidos por el monitorizador. De los 10790 paquetes transmitidos se han recibido 10065, es decir, se han perdido el 6,72% de los paquetes, debido al incremento de la tasa de transmisión.

De la misma manera que en la prueba anterior se comparará con el *ground truth*, que, para este caso, será tomado bajo las mismas condiciones, es decir, trasmitiendo a 1500 veces de la velocidad en el origen, lo que hace una tasa de 161,39 Mbps. También se comparan los resultados con la transmisión de paquetes a velocidad estándar para ver como evoluciona el aumento de transmisión de paquetes con la detección de paquetes.

Hay que tener en cuenta que, para este tipo de pruebas, tiene efecto el procesador del propio ordenador, así como la carga a la que esté sometido durante la realización de las pruebas. Esta anotación viene por la velocidad de *switching*, que dependiendo a la tasa a la que se reciba tráfico es muy probable que se descarte tráfico. De hecho, en este caso ya se aprecian esos descartes de tráfico.

Los resultados obtenidos se muestran en la figura 5-16, a continuación:

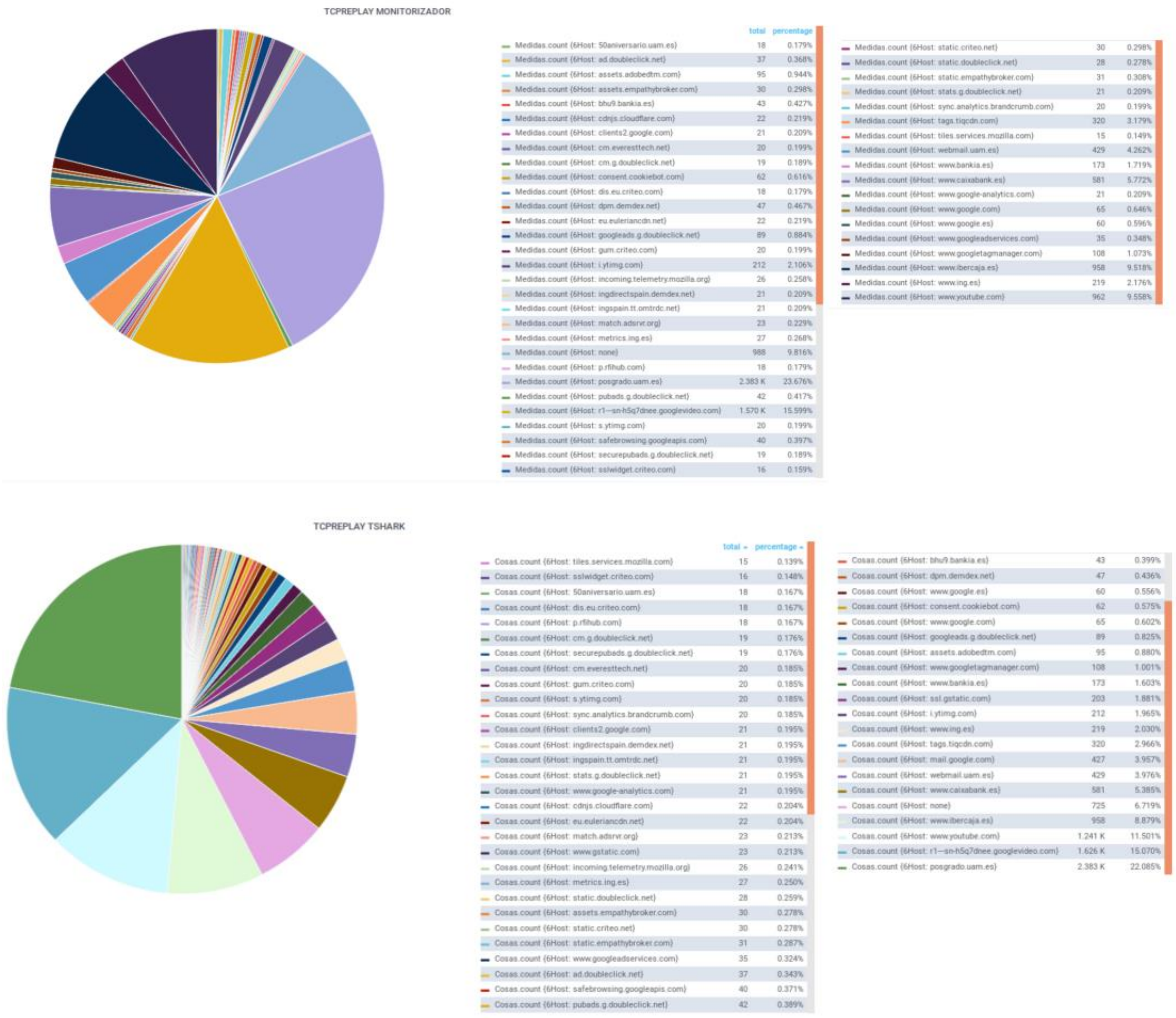


Figura 5-16: Porcentaje visitas HTTPs a hosts para 161,39 Mbps

En vista de las gráficas obtenidas se puede apreciar cómo se ha producido una disminución en la detección de host, debido a que, se han extraviado los paquetes que contenían el campo *Client Hello* y se han catalogado como none y porque hay una cantidad de paquetes que no se han podido procesar. De manera más detallada, a velocidad estándar se han identificado 51 hosts distintos, en este caso se han detectado 48 hosts.

En cuanto a los flujos que se han catalogado dentro del campo none, a velocidad estándar era un 6% del tráfico total, mientras que en esta ocasión ha incrementado al 9,8%. La degradación en la detección no es muy elevada, está por debajo del 15%, por lo que se puede decir que la tasa de detección es buena a 161,39 Mbps. El aumento de la velocidad de trasmisión no afecta a la capacidad de detección de thsark.

La tabla resumen que unifica los resultados sobre los errores relativos cometidos tras el aumento en la velocidad de transmisión se muestra a continuación:

Host	Error paquetes	Error Bytes
50aniversario.uam.es	0	0
ad.doubleclick.net	0	0
assets.adobedtm.com	0	0
assets.empathybroker.com	0	0
bhu9.bankia.es	0	0
cdnjs.cloudflare.com	0	0
clients2.google.com	0	0
cm.everesttech.net	0	0
cm.g.doubleclick.net	0	0
consent.cookiebot.com	0	0
dis.eu.criteo.com	0	0
dpm.demdex.net	0	0
eu.euleriancdn.net	0	0
googleads.g.doubleclick.net	0	0
gum.criteo.com	0	0
i.ytimg.com	0	0
incoming.telemetry.mozilla.org	0	0
ingdirectspain.demdex.net	0	0
ingspain.tt.omtrdc.net	0	0
mail.google.com	X	X
match.adsrvr.org	0	0
metrics.ing.es	0	0
p.rfihub.com	0	0
posgrado.uam.es	0	0
pubads.g.doubleclick.net	0	0
r1---sn-h5q7dnee.googlevideo.com	0,034440344	0,049028677
s.ytimg.com	0	0
safebrowsing.googleapis.com	0	0
securepubads.g.doubleclick.net	0	0
ssl.gstatic.com	X	X
sslwidget.criteo.com	0	0
static.criteo.net	0	0
static.doubleclick.net	0	0
static.empathybroker.com	0	0
stats.g.doubleclick.net	0	0
sync.analytics.brandcrumb.com	0	0
tags.tiqcdn.com	0	0
tiles.services.mozilla.com	0	0
webmail.uam.es	0	0
www.bankia.es	0	0
www.caixabank.es	0	0
www.google-analytics.com	0	0
www.google.com	0	0
www.google.es	0	0
www.googleadservices.com	0	0
www.googletagmanager.com	0	0
www.gstatic.com	X	X
www.ibercaja.es	0	0
www.ing.es	0	0
www.youtube.com	0,224818695	0,420805369

Tabla 5-6: Tabla resumen prueba HTTPs a 161,39 Mbps

De la anterior tabla se pueden extraer dos conclusiones, la primera de ellas es que hay tres hosts que no se han capturado paquetes. En realidad, es muy posible que, sí se hayan capturado paquetes de esos hosts, el problema reside en que el paquete que contiene la información relevante es el que no ha sido capturado, entonces todos los paquetes de ese flujo ya no se catalogan como deberían. La segunda conclusión es que con esta tasa ya empieza a bajar un poco el rendimiento del monitorizador aun siendo aceptable.

El resumen de la transmisión de los paquetes en la red virtualizada es:

Figura 5-17: Resumen transmisión con *tcpreplay* a 161,39 Mbps para HTTPs

Se han transmitido 10790 paquetes que suman un total de 6557495 Bytes y la duración de la inyección de tráfico es de 0.31 segundos. La tasa es de 161,39 Mbps y alrededor de 34806 paquetes por segundo.

5.3.3.3 Tráfico transmitido a 217,52 Mbps

En esta prueba se ha dado el salto de velocidad 1500 (161,39 Mbps) a velocidad por 2000 (217,52 Mbps) ya que en los pasos intermedios realizados no se detectaba variación notable, los resultados eran similares y no se podían extraer conclusiones de interés. Este paso intermedio se ha realizado para el ver el progreso de la perdida de paquetes antes de llegar al máximo de velocidad de transmisión. Al final del apartado se muestra un gráfico con la progresión de la pérdida de paquetes a medida que aumentaba la velocidad de transmisión.

De la misma manera que en los casos anteriores se comparan los resultados obtenidos con los extraídos bajo condiciones estándar y a su vez con los datos obtenidos con tshark.

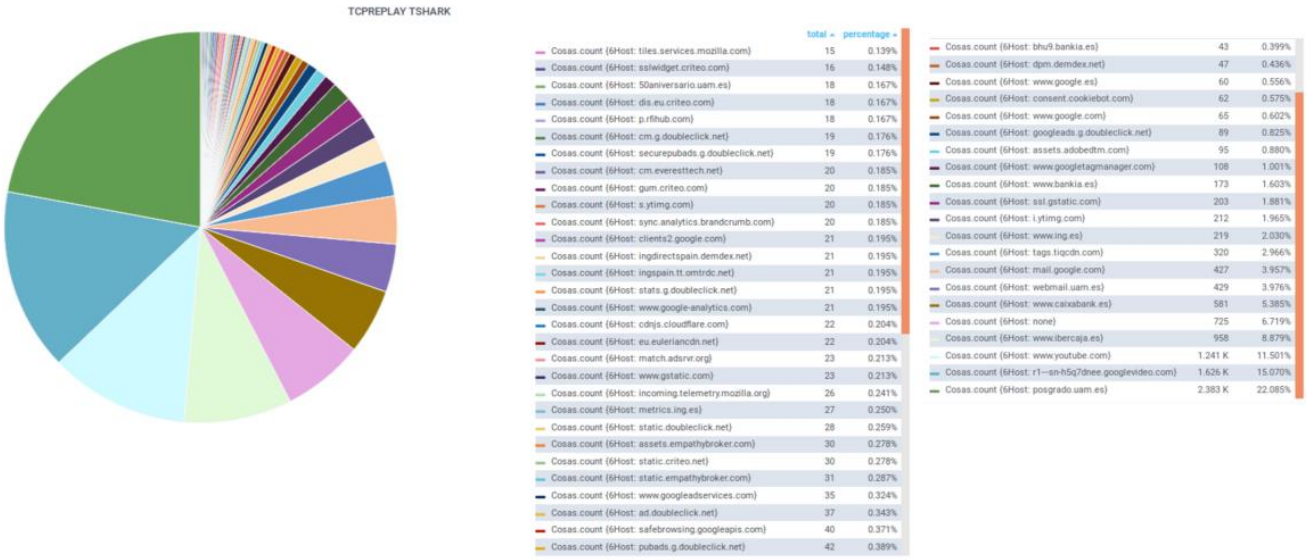


Figura 5-18: Porcentaje visitas HTTPs a hosts a 217,52 Mbps

A raíz de las gráficas obtenidas se puede observar cómo se ha producido un descenso en el porcentaje de detección de host, debido a que, se han perdido los paquetes que contenían el campo host y se han catalogado como none y porque hay una cantidad de paquetes que no se han podido procesar. En base a los datos concretos, a velocidad estándar se han identificado 51 hosts distintos, en este caso se han detectado 42 hosts.

En cuanto a los flujos que se han catalogado dentro del campo none, a velocidad estándar era un 6% del tráfico total, en el caso de 161,39 Mbps se ha incrementado al 9,8% y en este caso, el tráfico catalogado como none asciende al 12,78%. La degradación en la detección empieza a ser considerable, está por debajo del 15% de tráfico no catalogado, sin contar los paquetes que se han perdido y ni siquiera se han podido procesar. El aumento de la velocidad de trasmisión no afecta a la capacidad de detección de thsark, puesto que las gráficas obtenidas son iguales que a velocidad estándar.

A continuación, se muestra la tabla resumen del error relativo cometido en términos de paquetes y *bytes* con tasa de transmisión de 217,52 Mbps:

Host	Error paquetes	Error Bytes
50aniversario.uam.es	0	0
ad.doubleclick.net	0	0
assets.adobedtm.com	0	0
assets.empathybroker.com	0,966666667	0,977022222
bhu9.bankia.es	0	0
cdnjs.cloudflare.com	0	0
clients2.google.com	0	0
cm.everesttech.net	X	X
cm.g.doubleclick.net	0	0
consent.cookiebot.com	0	0
dis.eu.criteo.com	0	0
dpm.demdex.net	0	0
eu.eleriancdn.net	0	0
googleads.g.doubleclick.net	0	0
gum.criteo.com	0,15	0,217391304
i.ytimg.com	0	0
incoming.telemetry.mozilla.org	0	0
ingdirectspain.demdex.net	X	X
ingspain.tt.omtrdc.net	X	X
mail.google.com	X	X
match.adsrvr.org	0	0
metrics.ing.es	X	X
p.rfihub.com	0	0
posgrado.uam.es	0	0
pubads.g.doubleclick.net	0	0
r1---sn-h5q7dnee.googlevideo.com	0,330258303	0,43293247
s.ytimg.com	0	0
safebrowsing.googleapis.com	0	0
securepubads.g.doubleclick.net	0	0
ssl.gstatic.com	X	X
sslwidget.criteo.com	X	X
static.criteo.net	0	0
static.doubleclick.net	0	0
static.empathybroker.com	X	X
stats.g.doubleclick.net	0	0
sync.analytics.brandcrumb.com	0	0
tags.tiqcdn.com	0,4875	0,612535613
tiles.services.mozilla.com	0	0
webmail.uam.es	0	0
www.bankia.es	0	0
www.caixabank.es	0	0
www.google-analytics.com	0	0
www.google.com	0	0
www.google.es	0	0
www.googleadservices.com	0	0
www.googletagmanager.com	0,222222222	0,242741935
www.gstatic.com	X	X
www.ibercaja.es	0	0
www.ing.es	0,899543379	0,972027972
www.youtube.com	0,228041902	0,425503356

Tabla 5-7: Tabla resumen prueba HTTPs a 217,52 Mbps

En esta prueba la degradación en la captura es mayor y ya no se identifican varios hosts, porque no se ha capturado el paquete *Client Hello*, durante la transmisión y no se ha podido catalogar el resto de paquetes correspondientes a esos flujos. Concretamente no se han detectado nueve hosts. Por otro lado, los hosts que si se han podido etiquetar ha aumentado el número de errores tanto en el número de paquetes como el número de bytes debido a que el monitorizador no puede procesar todo el tráfico a la velocidad que le está llegando y se producen descartes de paquetes.

El resumen de la transmisión de los paquetes a una velocidad 2000 veces la estándar es:

```
Actual: 10790 packets (6557495 bytes) sent in 0.23 seconds.          Rated: 28510848.0 bps, 217.52 Mbps, 46913.04 pps
Statistics for network device: h1-eth0
        Attempted packets:         10790
        Successful packets:        10790
        Failed packets:            0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
```

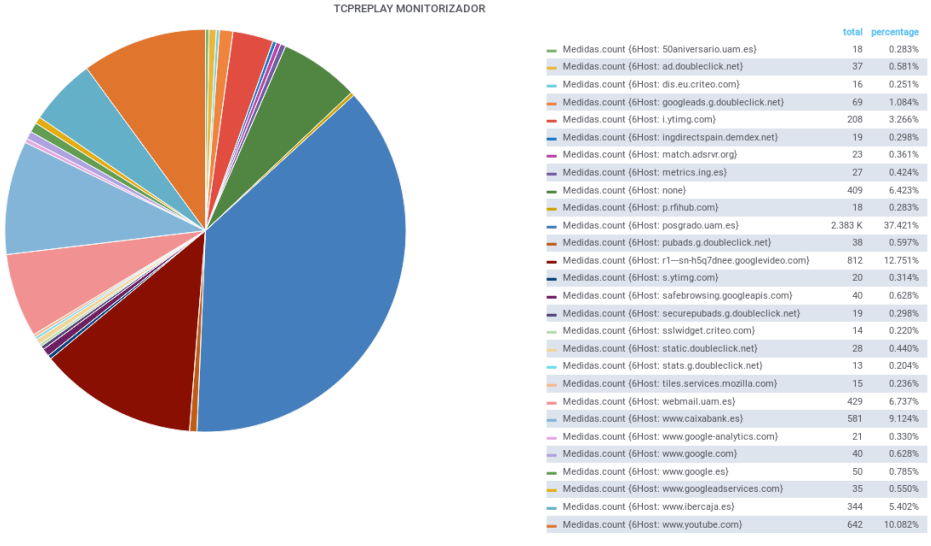
Figura 5-19: Resumen transmisión con *tcpreplay* a 217,52 Mbps para HTTPs

Se han transmitido 10790 paquetes que suman un total de 6557495 Bytes y la duración de la inyección de tráfico es de 0.23 segundos. La tasa es de 217,52 Mbps y alrededor de 46913 paquetes por segundo.

5.3.3.4 Tráfico transmitido al máximo de velocidad

En la última prueba de rendimiento para tráfico HTTPs, se retransmite los paquetes capturados con la opción –t de *tcpreplay* la cual permite emitir a la máxima velocidad posible. Esta prueba aportará visión de cómo se comportará el monitorizador implementado para redes de alta velocidad. De la misma manera que en las pruebas anteriores, se compararan los resultados para, al final del capítulo, poder obtener conclusiones fundamentadas. La tasa que se ha obtenido para esta prueba es de 714.71 Mbps

Bajo estas condiciones, el host emisor transmite 10790 paquetes hacia la red, pero el número de paquetes que el monitorizador es capaz de procesar son 6368. En este escenario el porcentaje de paquetes perdidos es del 41%. A continuación, se muestran los resultados de manera gráfica.



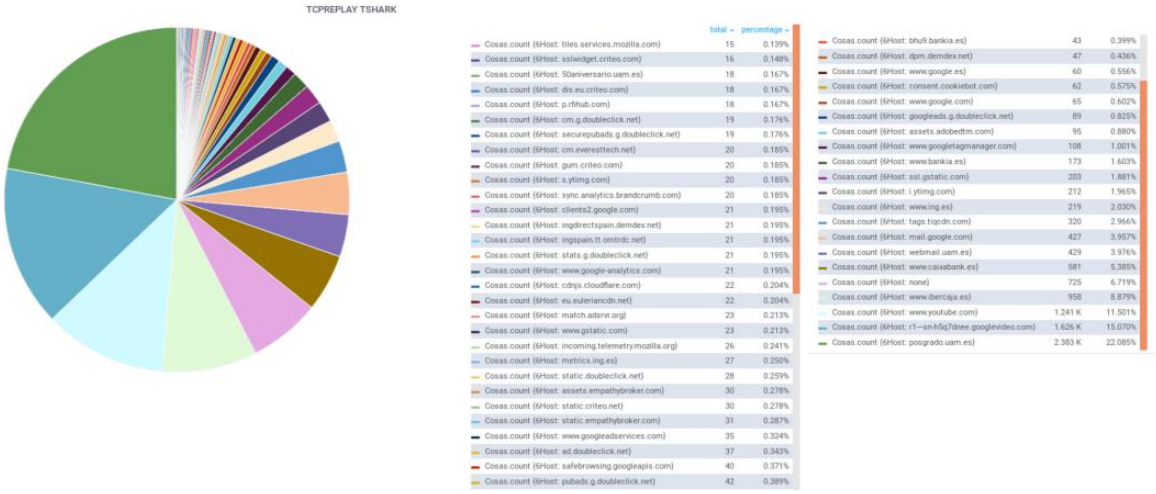


Figura 5-20: Porcentaje visitas HTTPs a hosts para máximo de velocidad

Se puede apreciar rápidamente observando la primera gráfica de la figura 5-20 como la detección de host con la tasa de transferencia no es buena. Concretamente, de los 51 host detectados con los que contábamos a velocidad de transmisión estándar, ahora, con el aumento al máximo de velocidad se han detectado 28 hosts.

En esta prueba el campo none abarca un 6% del tráfico total, pero no es muy relevante en esta ocasión debido a la gran cantidad de flujos que no se han podido analizar debido a la velocidad de transmisión tan elevada.

Se puede afirmar que a tasas tan altas de transmisión de datos el monitorizador implementado no funciona correctamente debido a la gran pérdida de información producida. En cambio, tshark si es capaz de procesar todo el tráfico. Para apreciar con mayor claridad este efecto se muestra la tabla resumen como en las pruebas anteriores donde se muestra el error relativo cometido en la captura con el monitorizador implementado con respecto a tshark:

Host	Error paquetes	Error Bytes
50aniversario.uam.es	0	0
ad.doubleclick.net	0	0
assets.adobedtm.com	X	X
assets.empathybroker.com	X	X
bhu9.bankia.es	X	X
cdnjs.cloudflare.com	X	X
clients2.google.com	X	X
cm.everesttech.net	X	X
cm.g.doubleclick.net	X	X
consent.cookiebot.com	X	X
dis.eu.criteo.com	0,111111111	0,163428571
dpm.demdex.net	X	X
eu.eulericancdn.net	X	X
googleads.g.doubleclick.net	0,224719101	0,31838565
gum.criteo.com	X	X
i.ytimg.com	X	X
incoming.telemetry.mozilla.org	X	X
ingdirectspain.demdex.net	0,095238095	0,139455782
ingspain.tt.omtrdc.net	X	X
mail.google.com	X	X
match.adsrvr.org	0	0
metrics.ing.es	0	0
none	0,435862069	264,9313725
p.rfihub.com	0	0
posgrado.uam.es	0	0
pubads.g.doubleclick.net	0,095238095	0,138763198
r1---sn-h5q7dnee.googlevideo.com	0,500615006	0,64199815
s.ytimg.com	0	0
safebrowsing.googleapis.com	0	0
securepubads.g.doubleclick.net	0	0
ssl.gstatic.com	X	X
sslwidget.criteo.com	0,125	0,170121951
static.criteo.net	X	X
static.doubleclick.net	0	0
static.empathybroker.com	X	X
stats.g.doubleclick.net	0,380952381	0,568020305
sync.analytics.brandcrumb.com	X	X
tags.tiqcdn.com	X	X
tiles.services.mozilla.com	0	0
webmail.uam.es	0	0
www.bankia.es	X	X
www.caixabank.es	0	0
www.google-analytics.com	0	0
www.google.com	0,384615385	0,575280899
www.google.es	0,166666667	0,245879121
www.googleadservices.com	0	0
www.googletagmanager.com	X	X
www.gstatic.com	X	X
www.ibercaja.es	0,64091858	0,678500986
www.ing.es	0,899543379	0,972027972
www.youtube.com	0,482675262	0,779865772

Tabla 5-8: Tabla resumen prueba HTTPs a 714,71 Mbps

Como cabía esperar se ha perdido gran cantidad de paquetes debido que la tasa a la que se estaban recibiendo era superior a la tasa de procesamiento del monitorizador. Para poder tratar de suavizar este efecto de pérdidas de paquetes se puede hacer un buffer donde esperen los paquetes a ser procesados y amortiguar los picos puntuales de tráfico

En esta captura se han recogido paquetes de host que en el caso anterior no se habían podido capturar. Este hecho depende del estado del controlador en el momento que van llegando los paquetes, no se puede predecir que paquetes se van a capturar y cuáles no.

El resumen de la tasa de transferencia de la captura es el siguiente:

Figura 5-21: Resumen transmisión con *tcpreplay* a 714,71 Mbps para HTTPs

Se han transmitido 10790 paquetes que suman un total de 6557495 Bytes y la duración de la inyección de tráfico es de 0.23 segundos. La tasa es de 714.71 Mbps y alrededor de 154142 paquetes por segundo.

5.3.3.5 Resultados globales

Para finalizar el apartado se muestran gráficas que relacionan el aumento de velocidad de transmisión con las pérdidas que se realizan en el monitorizador porque no es capaz de procesar todos los paquetes que está recibiendo a distintas velocidades. Las gráficas que se presentan son sobre el número de paquetes y número de bytes frente a la velocidad de transmisión. En primer lugar, se muestra la gráfica sobre el porcentaje de paquetes perdidos:

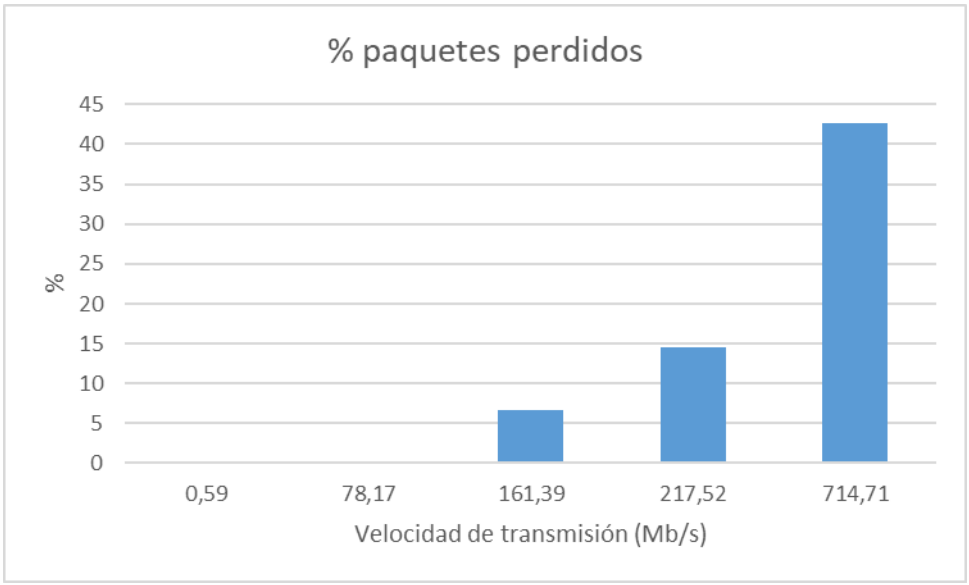


Figura 5-22: Resumen porcentaje paquetes perdidos para HTTPs

En la figura como se puede apreciar, como cabía esperar, que el rendimiento del monitorizador fuera decreciendo a medida que se incrementa la tasa de transmisión de datos por parte del emisor. Tanto es así, que en el máximo de velocidad el porcentaje de paquetes perdidos es del 41%, lo que hace evidente las carencias para identificar los flujos a grandes velocidades de transmisión por parte del monitorizador implementado.

A continuación, se muestra el porcentaje de pérdidas en bytes con las distintas velocidades de transmisión:

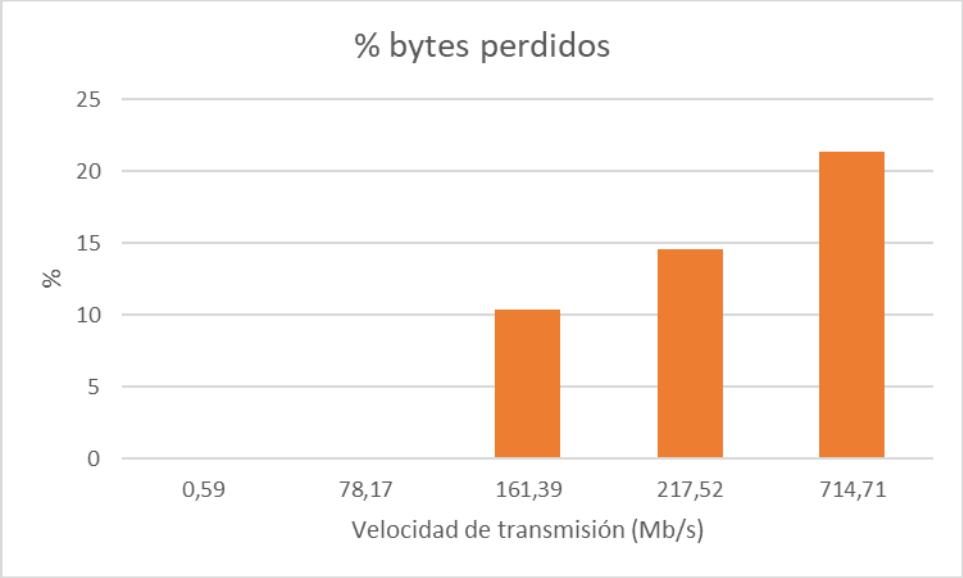


Figura 5-23: Resumen porcentaje bytes perdidos para HTTPs

En la gráfica anterior era posible anticipar el resultado, pero en este caso la pérdida es de paquetes es mucho más critica que, por ejemplo, para el caso de HTTP puesto que en esta ocasión si se pierde el paquete clave el resto de paquetes del flujo no se pueden catalogar y ya se cuentan como pérdidas.

5.4 Error relativo

En la presente prueba se muestra el error relativo que se ha cometido en las mediciones. Para ello se han concatenado las capturas con las que se ha trabajo en el apartado anterior y se han transmitido a la misma velocidad que fueron capturadas.

Una vez que se dispone de los datos capturados mediante el monitorizador, se organizan mediante flujos, de igual manera que en el apartado anterior. Cabe destacar, que para esta prueba no se ha tenido que modificar nada en los scripts implementados, únicamente se ha modificado el filtro de captura de paquetes del monitorizador de tal manera que procese todo el tráfico que reciba y, no como antes, que se centraba en cada tipo de tráfico. En lo que respecta a tshark, se ha modificado los parámetros de captura que imponían filtros al tráfico y se han añadido todos los campos de interés para la prueba. El comando resultante para obtener todos los campos de interés en tshark es el siguiente:

```
sudo tshark -i h3-eth0 -T fields -e frame.number -e frame.time -e
ip.src -e ip.dst -e tcp.srcport -e tcp.dstport -e udp.srcport -e
udp.dstport -e dns.a -e dns.resp.name -e udp.length -e
dns.flags.response -e http.host -e
ssl.handshake.extensions_server_name -e ssl.handshake.type -e
ssl.handshake.extensions_server_name_type -e tcp.len -E header=y
-E separator=, -E quote=d -E occurrence=f > salidatshark.csv
```

Para realizar la prueba se han seleccionado 30 flujos. Dichos flujos se obtienen mediante el monitorizador y mediante tshark, que actuara como *ground truth*. El campo que se va a comparar es el número de paquetes que se ha etiquetado dentro de ese flujo de manera correcta. Para el cálculo del error relativo se ha utilizado a siguiente formula.

$$Error = \frac{|\#Pckts_{Mon} - \#Pckts_{Tshark}|}{\#Pckts_{Tshark}}$$

Figura 5-24: Fórmula cálculo error relativo

La representación de los resultados se ha divido en dos gráficas, la primera de ellas muestra los resultados obtenidos para cada *sniffer* tras seleccionar los 30 flujos. Se representa en el eje horizontal el flujo etiquetado por host, en vez de por número de flujo, y en el eje vertical el número de paquetes capturado para cada flujo durante la prueba. La segunda gráfica muestra el error relativo calculado mediante la fórmula de la figura 5-24.

A continuación, se muestra la primera gráfica con los resultados extraídos de ambos monitorizadores sobre la diferencia de paquetes capturados:

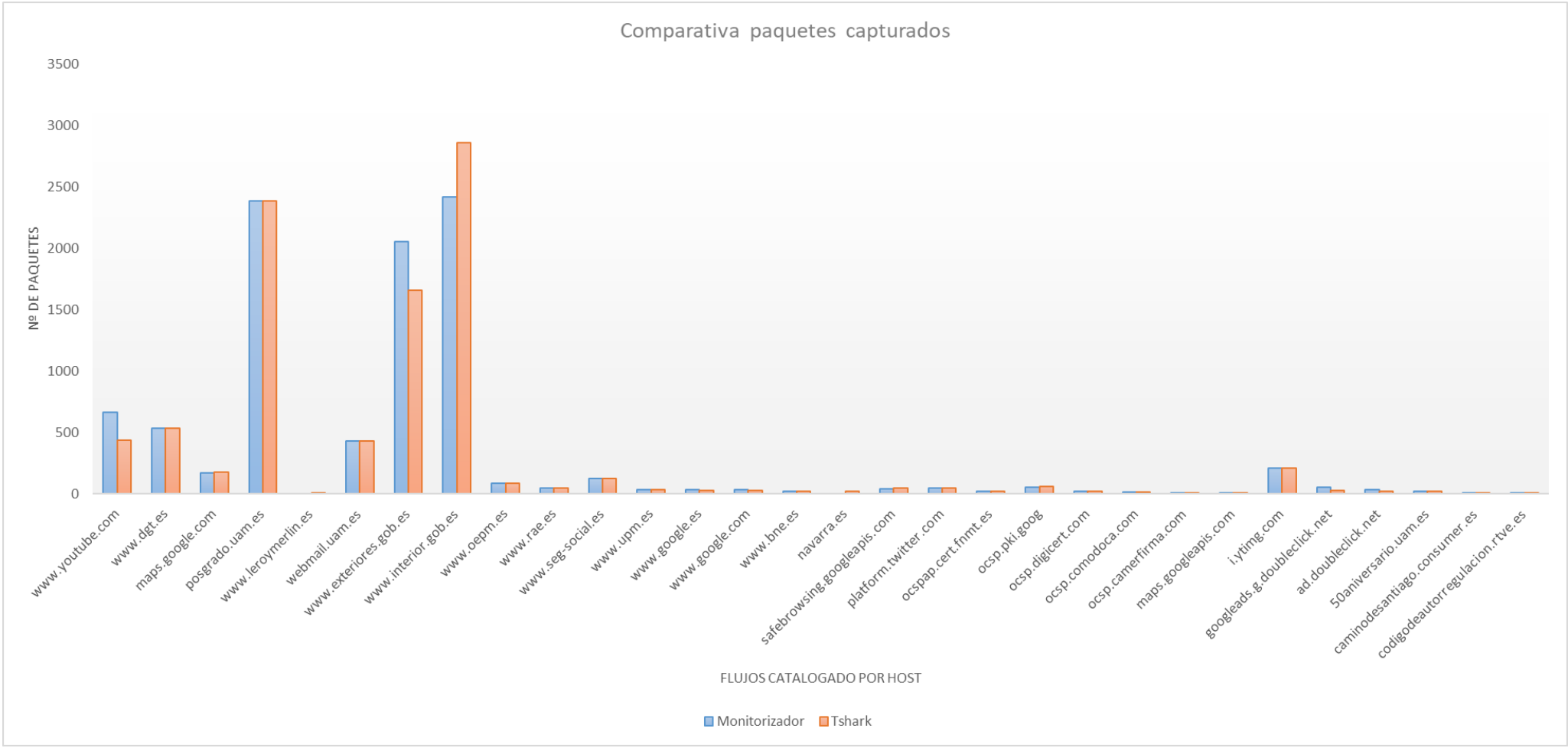


Figura 5-25: Comparativa paquetes capturados con monitorizador y tshark

En la gráfica superior se muestra el resultado de la captura de paquetes con el *sniffer* implementado y con tshark para cada flujo etiquetado. Se puede observar como ambos siguen una dinámica muy similar, pero en algunos puntos sí que difiere el número de paquetes obtenido mediante cada mecanismo de captura. La línea azul se corresponde con los resultados del *sniffer* implementado mientras que la línea naranja se corresponde con los resultados de tshark. Estas diferencias se harán más notables en la próxima gráfica que muestra el error relativo cometido para cada host detectado.

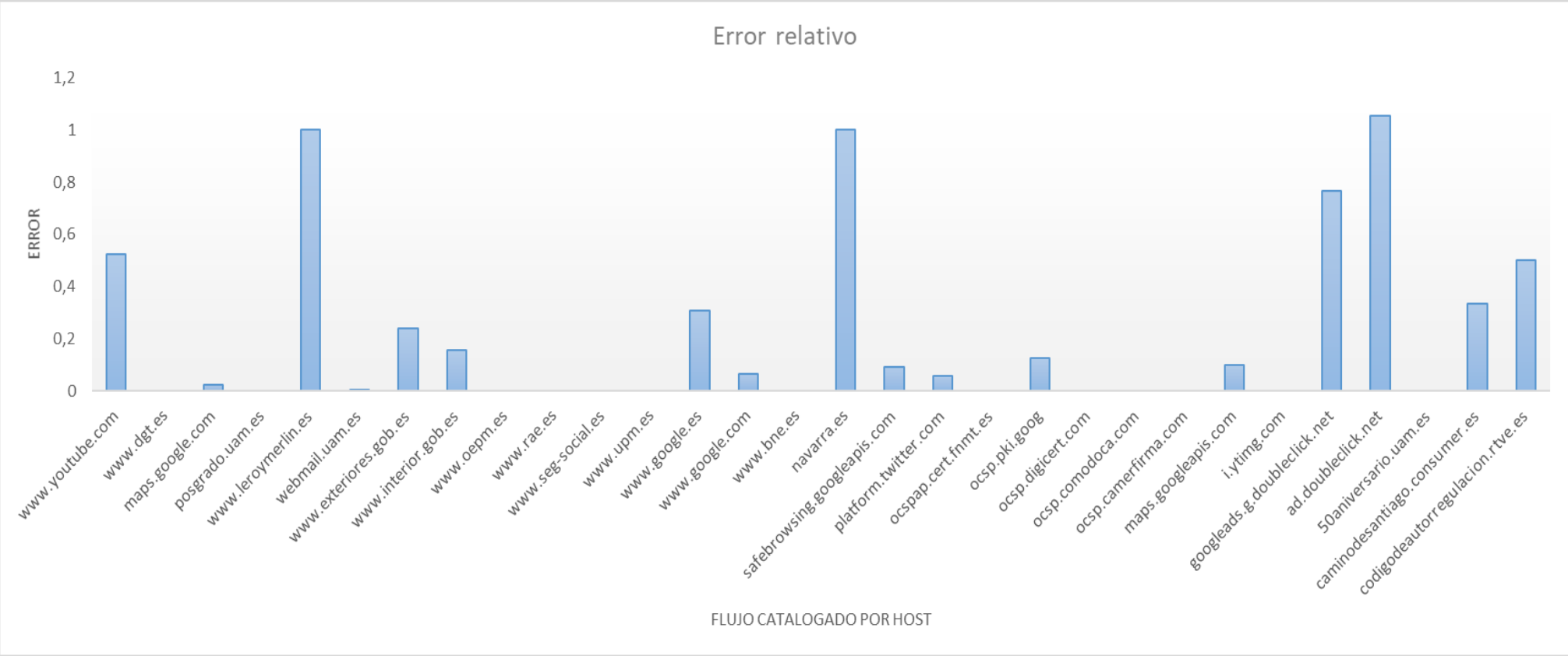


Figura 5-26: Gráfica de error relativo

En la gráfica 5-26 se muestra el error que se produce en la medición para cada flujo. Se puede observar algunos flujos en los que el error que se comete es elevado, en otros casos no existe error. En media el error cometido es del 0,2. Es un error relativo medio bajo y por lo tanto se puede concluir que el *sniffer* implementado ofrece buena tasa de detección de hosts.

5.5 *Conclusiones*

Para finalizar el apartado de pruebas se va a resumir y enunciar las conclusiones principales que se pueden extraer de este apartado.

El apartado de pruebas se ha abordado con la visión de lo más concreto a lo más general. De esa manera se comienza explicando cómo se obtienen los datos con los que son comparan los resultados obtenidos. En siguiente lugar, se ha realizado la prueba del rendimiento, la cual, tenía como finalidad poner a prueba el *sniffer* en cuanto a tasas de transferencia de datos y ver como de bueno era con las variaciones de velocidad de transmisión. Se ha detectado el umbral a partir el cual el rendimiento se ve degradado y deja de ser fiable.

De la primera prueba en base a los resultados obtenidos, se puede afirmar que el *sniffer* implementado es bueno a velocidades estándar de captura de los distintos tipos de tráfico. También se ha extraído que, en función del tipo de tráfico, debido al procesamiento de cada paquete, hay diferentes umbrales de detección.

En la segunda prueba se ha observado el error relativo cometido en las mediciones durante la primera prueba para cada flujo clasificado. Las conclusiones que se obtienen es que la tónica general es que se cometen pocos errores en la captura y catalogación de los paquetes, pero los flujos en los que se detecta errores dichos errores son notables.

De manera global, se puede afirmar que el *sniffer* es válido para identificar catalogar paquetes en redes *OpenFlow* hasta ciertos puntos de tasas de transferencia no muy elevadas. Se ha podido alcanzar esta conclusión gracias a las pruebas realizadas en este apartado.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

El objetivo de este trabajo era desarrollar un *framework* de monitorización para aplicaciones usando *OpenFlow*. Para ello ha sido necesario encontrar una plataforma en la que se ejecute *OpenFlow* y sobre la que se pueda simular una red sencilla para poder hacer pruebas y verificar el correcto funcionamiento.

Una vez que se obtuvo la base sobre la que se iba a sustenta la implementación, en el caso de este trabajo la opción fue Mininet, se procedió al estudio de los distintos elementos que tienen cabida en el desarrollo. Entre los elementos que conforman el trabajo se puede destacar el controlador, en nuestro caso RYU. Otra parte importante es el programa que se encarga de monitorizar el tráfico, que está desarrollado en C y basado en el contenido de [9]. El programa encargado de dar sentido a los datos extraídos mediante el monitorizador se ha implementado en Python sin estar basado en nada, se creó desde cero. Por último, una pieza importante es Grafana para poder visualizar y comprender el rendimiento del *sniffer*.

En el trabajo se ha repasado las distintas alternativas a *OpenFlow* para poner en contexto y mostrar la importancia que tiene el protocolo, así como también se ha hecho un breve resumen sobre algunos de los distintos controladores *open source* que se encuentran disponibles en el mercado en la actualidad.

Para finalizar y, una vez que se disponía de una visión global de todo el alcance del trabajo de fin de máster se han realizado un batería de pruebas con el fin de evaluar como de bueno y, si realmente es útil, el monitorizador implementado.

Las conclusiones que se obtienen del trabajo de fin de máster son, que el espectro de protocolos para redes definidas por software es amplío tanto dentro del entorno *open source* como en el ámbito de empresas privadas. Tras las indagaciones realizas se ha observado como *OpenFlow* se sitúa en una posición muy ventajosa por su versatilidad para múltiples tipos de entornos y fabricantes. Además, está en constante mejora y evolución. En lo que respecta a las controladoras existe un amplio espectro y cada uno tiene sus ventajas e inconvenientes dependiendo de la finalidad a que se vaya a dedicar. En nuestro caso RYU, está dentro de los mejores para el ámbito de la monitorización.

En último lugar se han evaluado una serie de características del *framework* implementado mediante una serie de pruebas. De las pruebas se puede concluir que el *framework* implementado presenta una buena respuesta a velocidades estándar con una tasa de detección e identificación de tráfico muy alta, acercándose mucho a la realidad. También con las pruebas realizadas se ha detectado el rango de trabajo en el que es útil y, en el siguiente apartado se propondrán una serie de mejoras para trabajos futuros para poder solventar las posibles carencias y mejorar el rendimiento a tasas de envío de tráfico más elevadas.

En lo que respecta a los errores cometidos en las medidas se detectan pocos errores, aunque en aquellos flujos en los que hay errores, estos errores son considerables.

En último lugar, para recapitular todo lo anteriormente mencionado, el *framework* implementado es bueno para identificar y catalogar el tráfico que atraviesa la red, así como para extraer información sobre el volumen de Bytes que están circulando por la red y con

qué flujo se corresponden. Se puede concluir que se ha alcanzado el objetivo del trabajo de fin de máster.

6.2 *Trabajo futuro*

Como líneas de trabajo futuro se van a dividir en dos grupos. El primer grupo de posibles ideas para trabajos futuros se corresponderá con la inclusión de mejoras para poder incrementar el rendimiento y la tasa de detección de los flujos. Y, en segundo lugar, se propondrán trabajos para ampliar el alcance del trabajo actual. Algunos ejemplos de este segundo bloque pueden ser probar con otros controladores que pueden ser útiles, o que al detectar cierto tipo de tráfico se apliquen cierto tipo de políticas para que sea tratado de una manera u otra.

Trabajos futuros para tratar de mejorar el rendimiento del trabajo actual:

- Optimizar las funciones que se encargan de recorrer el *payload* para encontrar los campos de interés, para cada tipo de tráfico.
- Establecer comunicación mediante sockets entre el *sniffer* y colector de flujos para evitar tener que escribir en disco, lo cual ralentiza el proceso.
- Optimizar el colector de la información que se recibe del *sniffer* para que trabaje a tasas de datos más altas.
- Implementar un búfer en el *sniffer* para mitigar las pérdidas de paquetes para altas tasas de transmisión de datos.
- Implementar todo el *framework* en único lenguaje de programación.
- Implementar el escenario para que trabaje a tiempo real y se actualicen los datos según se va recibiendo la información.
- Aumentar la capacidad de detección implementando búfer para procesar paquetes en redes de alta velocidad y que no sean descartados

Trabajos futuros para ampliar el alcance del trabajo actual:

- Añadir la catalogación de otro tipo de tráfico. Algunos ejemplos pueden ser SMTP, FTP entre otros.
- Extender la catalogación del tráfico TCP para que sea capaz de reconstruir paquetes en los que se produzca segmentación.
- Probar el desarrollo en un escenario real con OpenWRT.
- Realizar el estudio con distintos controladores OpenSource para detectar cual presenta mejor rendimiento para entornos de monitorización.
- Imponer reglas al tráfico en función de la naturaleza que sea. Algunas reglas pueden ser priorización o descarte del tráfico, entre otras.

Referencias

[1] McKeown, N. (2009). Software-defined networking. INFOCOM keynote talk, 17(2), 30-32.

[2] Bradley, W., Maher, D., & Boccon-Gibod, G. (2012). U.S. Patent No. 8,234,387. Washington, DC: U.S. Patent and Trademark Office.

[3] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J. & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

[4] Enns, R., Bjorklund, M., & Schoenwaelder, J. (2006). NETCONF configuration protocol. RFC 4741, December.

[5] Narisetty, R., Dane, L., Malishevskiy, A., Gurkan, D., Bailey, S., Narayan, S., & Mysore, S. (2013, March). OpenFlow configuration protocol: implementation for the of management plane. In 2013 second GENI research and educational experiment workshop (pp. 66-67). IEEE.

[6] De Oliveira, R. L. S., Schweitzer, C. M., Shinoda, A. A., & Prete, L. R. (2014, June). Using mininet for emulation and prototyping software-defined networks. In 2014 IEEE Colombian Conference on Communications and Computing (COLCOM) (pp. 1-6). Ieee.

[7] Man page tshark. https://www.wireshark.org/docs/man-pages/tshark.html

[8] Kloth, R. J., Edsall, T. J., Ghosh, K. K., Rastogi, G., Dutt, D. G., & Cressa, M. (2009). U.S. Patent No. 7,474,666. Washington, DC: U.S. Patent and Trademark Office.

[9] Programming with pcap. https://www.tcpdump.org/pcap.html

[10] Grafana. https://grafana.com/

[11] Taher, A. (2014). Testing of floodlight controller with mininet in sdn topology. ScienceRise, (5 (2)), 68-73.

[12] Khattak, Z. K., Awais, M., & Iqbal, A. (2014, December). Performance evaluation of OpenDaylight SDN controller. In 2014 20th IEEE international conference on parallel and distributed systems (ICPADS) (pp. 671-676). IEEE.

[13] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., ... & Parulkar, G. (2014, August). ONOS: towards an open, distributed SDN OS. In Proceedings of the third workshop on Hot topics in software defined networking (pp. 1-6).

[14] NOX Controller. https://github.com/noxrepo/nox. POX controller. https://github.com/noxrepo/pox

[15] RYU controller. https://ryu.readthedocs.io/en/latest/

[16] Nuage VSC controller. https://www.nuagenetworks.net/course/vcs-fundamentals/

[17] VortiQa controller. https://www.nxp.com/design/software/embedded-software/vortiqa-software-for-networking:VORTIQA

[18] Unifi controller. https://www.ui.com/software/

[19] SDN-Architecture Northbound-southbound https://en.wikipedia.org/wiki/File:Software_Defined_Networking_System_Overview.svg

[20] Open Network Foundation. https://www.opennetworking.org/

[21] Flow-Tables Open Flow. https://www.sdxcentral.com/networking/sdn/definitions/what-is-OpenFlow/

[22] Enns, R., Bjorklund, M., Schoenwaelder, J., & Bierman, A. (2011). Network configuration protocol (NETCONF).

[23] Bierman, A., Bjorklund, M., Watsen, K., & Fernando, R. (2017). RESTCONF protocol. IETF RFC 8040.
[24] OF-CONFIG documento ONF https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/OpenFlow-config/of-config-1.2.pdf
[25] Pfaff, B., & Davie, B. (2013). Rfc 7047: The open vswitch database management protocol. Internet Engineering Task Force (IETF).
[26] VXLAN with OVSDB, https://techhub.hpe.com/eginfolib/networking/docs/switches/7500/5200-1957a_vxlan_cg/content/495503752.htm
[27] Repositorio Trema. https://github.com/trema/trema-edge
[28] Yu, M., Jose, L., & Miao, R. (2013). Software Defined Traffic Measurement with OpenSketch. In Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13) (pp. 29-42).
[29] Switch OpenFlow. https://www.opennetworking.org/wp-content/uploads/2014/10/OpenFlow-switch-v1.4.1.pdf
[30] Yassine, A., Rahimi, H., & Shirmohammadi, S. (2015). Software defined network traffic measurement: Current trends and challenges. IEEE Instrumentation & Measurement Magazine, 18(2), 42-50.
[31] Van Adrichem, N. L., Doerr, C., & Kuipers, F. A. (2014, May). Opennetmon: Network monitoring in OpenFlow software-defined networks. In 2014 IEEE Network Operations and Management Symposium (NOMS) (pp. 1-8). IEEE.
[32] Malboubi, M., Wang, L., Chuah, C. N., & Sharma, P. (2014, April). Intelligent sdn based traffic (de) aggregation and measurement paradigm (istamp). In IEEE INFOCOM 2014-IEEE Conference on Computer Communications (pp. 934-942). IEEE.
[33] Suárez-Varela, J., & Barlet-Ros, P. (2017). Reinventing netflow for OpenFlow software-defined networks. arXiv preprint arXiv:1702.06803.
[34] Yoon, S., Ha, T., Kim, S., & Lim, H. (2017). Scalable traffic sampling using centrality measure on software-defined networks. IEEE Communications Magazine, 55(7), 43-49.
[35] Tootoonchian, A., Ghobadi, M., & Ganjali, Y. (2010, April). OpenTM: traffic matrix estimator for OpenFlow networks. In International Conference on Passive and Active Network Measurement (pp. 201-210). Springer, Berlin, Heidelberg.
[36] Chowdhury, S. R., Bari, M. F., Ahmed, R., & Boutaba, R. (2014, May). Payless: A low cost network-monitoring *framework* for software defined networks. In 2014 IEEE Network Operations and Management Symposium (NOMS) (pp. 1-9). IEEE.
[37] Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., & Fonseca, R. (2014). Planck: Millisecond-scale monitoring and control for commodity networks. ACM SIGCOMM Computer Communication Review, 44(4), 407-418.
[38] Suh, J., Kwon, T. T., Dixon, C., Felter, W., & Carter, J. (2014, June). Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In 2014 IEEE 34th International Conference on Distributed Computing Systems (pp. 228-237). IEEE.
[39] Khondoker, R., Zaalouk, A., Marx, R., & Bayarou, K. (2014, January). Feature-based comparison and selection of Software Defined Networking (SDN) controllers. In 2014 World Congress on Computer Applications and Information Systems (WCCAIS) (pp. 1-7). IEEE.
[40] Software Ns-3. https://www.nsnam.org/

[41] Repositorio con el código implementado y utilizado en el trabajo de fin de Máster. https://github.com/Rsscoseno/TFM_MASTER_MUIT
[42] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). RFC 2616: Hypertext transfer protocol–HTTP/1.1, June 1999. Status: Standards Track, 1(11), 1829-1841.
[43] Mockapetris, P. (2004). RFC 1035—Domain names—implementation and specification, November 1987. URL http://www. ietf. org/rfc/rfc1035. txt.
[44] Rescorla, E. (2000). HTTP Over TLS (RFC 2818). Internet Engineering Task Force.

Glosario

API	Application Programming Interface
ONF	Open Network Foundation
OF-CONFIG	OpenFlow Configuration
DNS	Domain Name System
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
SDN	Software Defined Network
VXLAN	Virtual Extensible Local Area Network
VTEP	VXLAN Tunnel End Point
SPAN	Switched Port Analyzer
QoS	Quality of Service
XML	Extensible Markup Language
TLS	Transport Layer Security
SSH	Secure Shell
SNMP	Simple Network Management Protocol
RFC	Request for Comments
TM	Traffic Matrix
RTT	Round-Trip Time
TCAM	Ternary Content-Addressable Memory
C	Lenguaje de programación C
IP	Internet Protocol
YANG	Yet Another Next Generation (Modelado de datos)
JSON	JavaScript Object Notation
RPC	Remote Procedure Call

Anexos

A Manual de instalación

En el trabajo de Fin de Máster se ha utilizado para el desarrollo Linux, concretamente Ubuntu, versión 16.04. Todos los pasos para la instalación del entorno son para ese sistema operativo y esa versión. Se va a suponer que el entorno en el que se trabaja es nuevo

El primer elemento a instalar es Mininet. Para la instalación de Mininet se tienen que seguir los siguientes pasos. Existen distintas maneras, aquí se va a mostrar la que se realizó para este proyecto.

Pasos para instalar Mininet:

- Obtener el código fuente de github:
  ```
  git clone git://github.com/mininet/mininet
  ```

- Una vez que se dispone del repositorio se procede a instalar Mininet:
  ```
  mininet/util/install.sh [options]
  ```

- Para verificar que la instalación se puede comprobar mediante:
  ```
  sudo mn --test pingall
  ```

En siguiente lugar es necesario instalar los compiladores e intérpretes necesarios:

- Compilador de C
  ```
  sudo apt install gcc
  ```

- Interprete de Python. En nuestro caso es Python 3.
  ```
  sudo apt-get install python3.6
  ```

Para continuar es necesario instalar Grafana e influxdb

- Instalación Grafana
  ```
  sudo apt-get install -y gnupg2 curl
  curl https://packages.grafana.com/gpg.key | sudo apt-key add –
  sudo add-apt-repository "deb
  https://packages.grafana.com/oss/deb stable main"
  sudo apt-get -y install grafana
  sudo systemctl start grafana-server
  sudo systemctl enable grafana-server
  ```

- Instalación Influxdb
  ```
  curl -sL https://repos.influxdata.com/influxdb.key |
  sudo apt-key add -source /etc/lsb-release
  ```

```
echo "deb https://repos.influxdata.com/${DISTRIB_ID,,}
${DISTRIB_CODENAME} stable" | sudo tee
/etc/apt/sources.list.d/influxdb.list
sudo apt-get update && sudo apt-get install influxdb
sudo service influxdb start
```

En último lugar es necesario instalar tshark

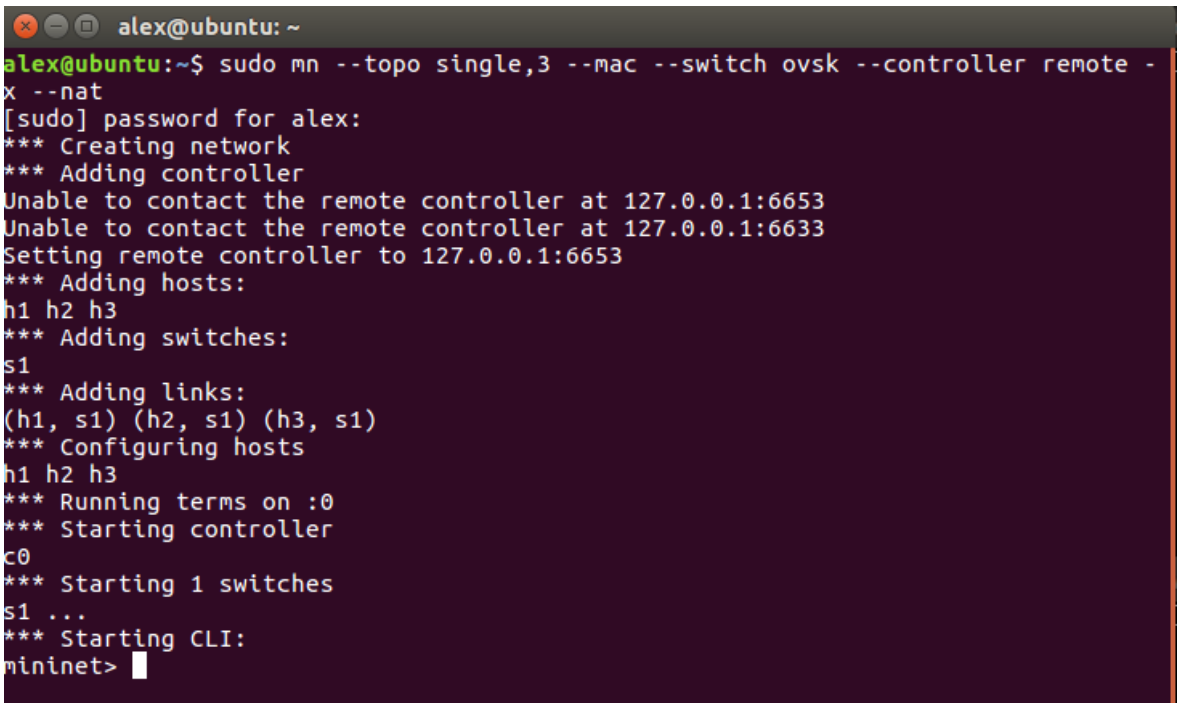
- Instalación tshark
  ```
  sudo apt-get install -y tshark
  ```

B Ejecución del escenario

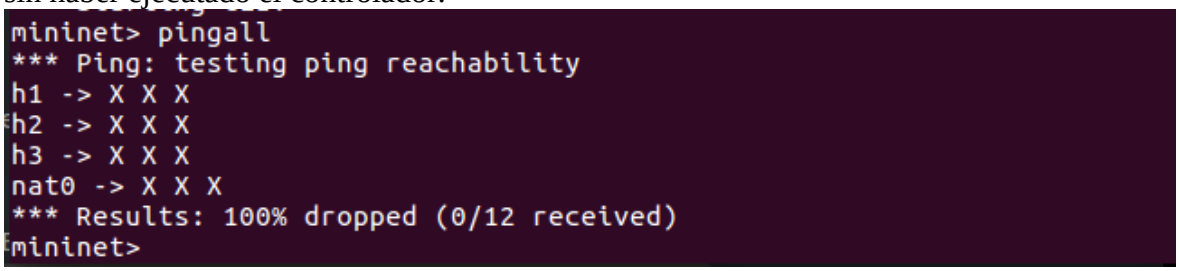
Para finalizar se muestra el proceso de una ejecución del escenario completo.

1º Ejecutar el escenario de mininet:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote -x --nat
```



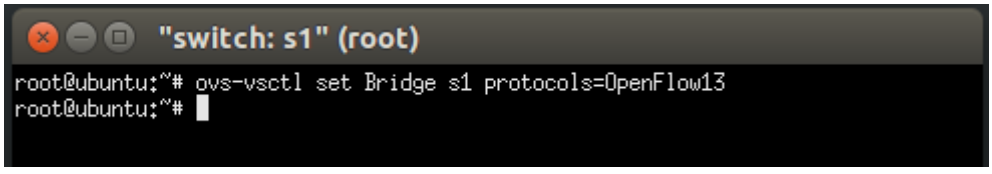
Hacemos un ping sin haber indicado la versión de OpenFlow que va a ejecutar el switch y sin haber ejecutado el controlador.



En siguiente lugar indicamos en la consola del switch la versión de OpenFlow que va a ejecutar y en el controlador ejecutamo el comando para arrancar el controlador de la red.

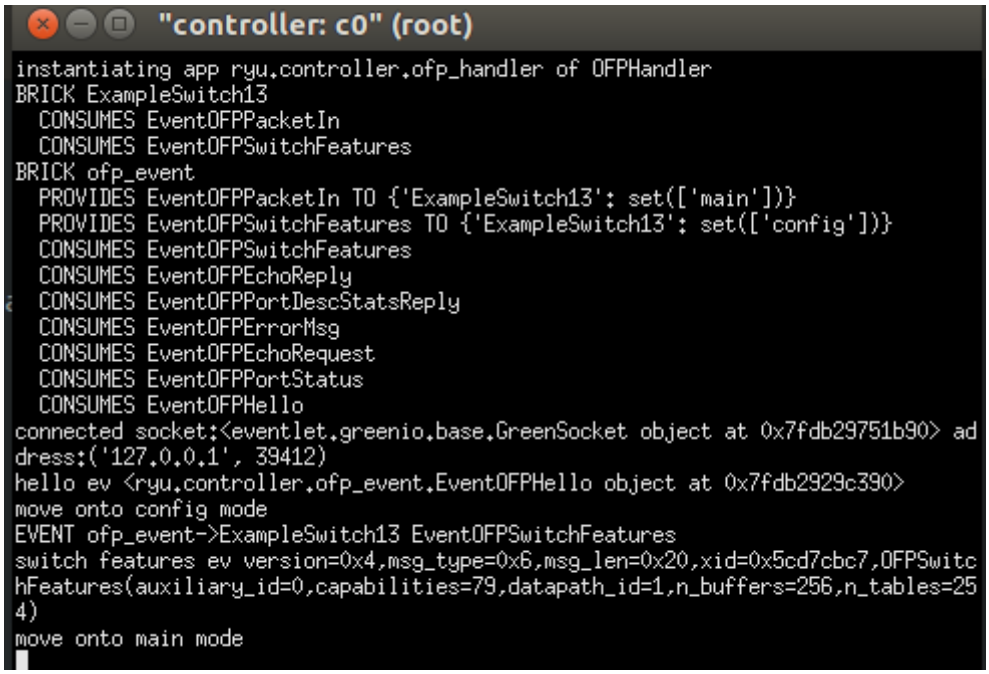
Versión1.3 de OpenFlow:

```
ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

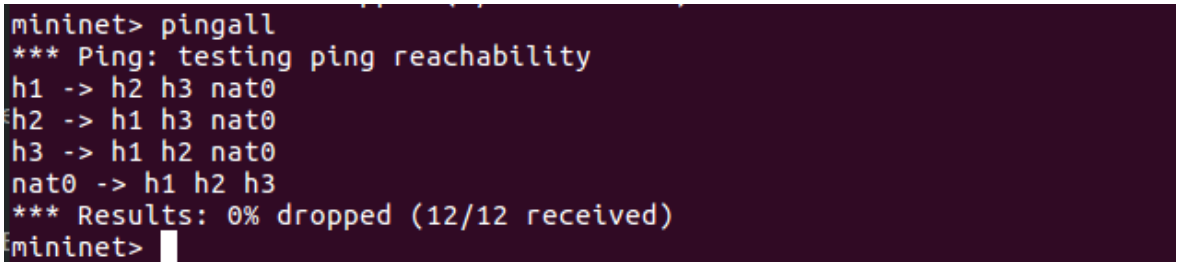


Inicio del controlador con la modificación del puerto SPAN:

```
ryu-manager --verbose SPAN.py
```



Una vez que esta todo funcionando podemos realizar el ping entre todos los nodos para verificar la conectividad:



Se puede observar como ahora si que todos los nodos son alcanzables ya ademas ya estan todas las tablas de reenvío almacenadas.

En siguiente lugar es necesario ejectuar el monitorizar en HMON (host 3) y comenzar a transmitir el tráfico desde host 1.

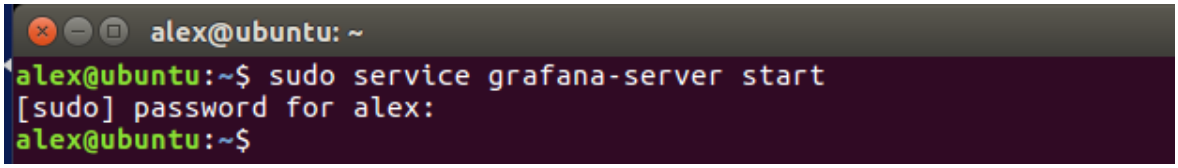
Ejecución en host 3:

Ejecución en host 1:

```
sudo tcpreplay --intf1=h1-eth0  24PAginasVisitadas-53+80+443.pcap
```

Una vez que se finaliza la captura de los paquetes se procede a ejecutar el colector en el controlador, el cual lee los datos recibidos , los agrupa por flujos y lo escribe en la base de datos de influx para luego ser representados con Grafana.

Arrancamos el servicio de Grafana:



Ejecución del colector:

Obtención del grountruth:

```
sudo tshark -i ens33 -T fields -e frame.number -e frame.time -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport -e http.host -e tcp.len -E header=y -E separator=, -E quote=d -E occurrence=f > archivo_salida.csv
```

Obtención de HTTP:

```
sudo tshark -i h3-eth0 -f "tcp port 80" -T fields -e frame.number -e frame.time -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport -e http.host -e tcp.len -E header=y -E separator=, -E quote=d -E occurrence=f-c 300000 > Port80.csv
```

Obtención de DNS:

```
sudo tshark -i h3-eth0 -T fields -e frame.number -e frame.time -e ip.src -e ip.dst -e udp.srcport -e udp.dstport -e dns.a -e
```

```
dns.resp.name -e udp.length -e dns.a -e dns.afsdb.subtype -e
dns.flags.response -E header=y -E separator=, -E quote=d -E
occurrence=f -c 1080 > salidatshark.csv
```

Obtención de HTTPs:

```
sudo tshark -i h3-eth0 -f "tcp port 443" -T fields –e frame.number
-e frame.time -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport -e
ssl.handshake.extensions_server_name -e tcp.len -e
ssl.handshake.type -e ssl.handshake.extensions_server_name_type -E
header=y -E separator=, -E quote=d -E occurrence=f -c 10790 >
salidatshark.csv
```

Ejemplo de matcheo de campos según las reglas indicadas para tomar acciones:

```
match = parser.OFPMatch(
     in_port=1,
     eth_type=0x86dd,
     ipv4_src=(‘192.168.1.43’),
     ipv4_dst=(‘54.240.186.96’)
 # query
 if 'ipv4_src' or ipv4_dst in match:
     lista.append(host)
```

Instalación de reglas en el switch mediante CURL y lo aprendido de la red. Se muestra en formato JSON:

```
curl -X POST -d '{
    "dpid": 1,
    "cookie": 1,
    "cookie_mask": 1,
    "table_id": 0,
    "idle_timeout": 0,
    "hard_timeout": 0,
    "priority": 1,
    "flags": 1,
    "match":{
        "ipv4_src":192.168.1.43,
        "ipv4_dst": 54.240.186.96,
        "tcp_src": 36944,
        "tcp_dst": 443,

    },
    "actions":[
        {
            "type":"OUTPUT",
            "port": 2,
```

```
            "port": 3
        }
    ]
}'http://localhost:8080/stats/flowentry/add
```